VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Knihovna pro práci s konečnými automaty

Finite Automata Library

Adam Kiovský

Bakalářská práce

Vedoucí práce: doc. Mgr. Jiří Dvorský, Ph.D.

Ostrava, 2021

Abstrakt

Cílem této bakalářské práce je navrhnout a implementovat funkce pro komplexní práci s konečnými automaty. Tyto funkce jsou implementovány v jedné společné knihovně. Celá knihovna byla napsána v jazyce C#. Součást knihovny jsou také algoritmy pro převod regulárních jazyků na konečný automat. Výstupem programu jsou řetězce, díky kterým si uživatel může nechat vygenerovat konkrétní automat ve vizualizačním programu GraphViz.

Klíčová slova

Bakalářská práce, Konečné automaty, Regulární výraz, Regulární gramatika, C#, .NET

Abstract

The aim of this bachelor thesis is to design and implement functions for complex work with finite automata. Those functions are implemented in the one common library. Whole library is written in C# language. A part of the library are algorithms for transfering regular languages into finite automaton. An output is a string, which can be used for generating images of automaton in GraphViz program.

Keywords

Bachelor thesis, Finite automata, Regular expression, Regular grammar, C#, .NET

Poděkování

Na tomto místě bych chtěl poděkovat panu Ing. Lukáši Tomaszkovi za cenné rady, panu Ing. Martinu Kotovi, Ph.D. za odborné rady a panu doc. Mgr. Jiřímu Dvorskému, Ph.D. za cenné i odborné rady, odborné vedení a trpělivost.

Obsah

Seznam použitých zkratek a symbolů

KA	–	Konečný automat
DKA	–	Deterministický konečný automat
NKA	–	Nedeterministický konečný automat
ZNKA	–	Zobecněný nedeterministický konečný Automat
RV	–	Regulární výraz
BNF	–	Backus-Naurova forma

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Cílem bakalářské práce je přiblížit čtenářům problematiku konečných automatů a také práci s nimi, jak po teoretické stránce, tak po stránce praktické. Tato práce může být nápomocna studentům vysokých škol, zpravidla technických oborů. I v profesním životě mají konečné automaty své využití, například při programování her.

Práce je rozdělena do tří částí. První část, kapitoly 2, 3 a 4, obsahují teoretický popis konečných automatů a jejich základní rozdělení. Dále jsou v této části obsažené teoretické popisy algoritmů, které v knihovně využívám. Druhá část, kapitola 5, je zaměřena na implementaci knihovny. Třetí část, kapitola 6, obsahuje testování funkčnosti algoritmů a jejich výstupy.

Použitá literatura

O konečných automatech existuje spousta historické i novodobé literatury. Každý zdroj pochopitelně definuje automaty a operace s ním spojené trochu jinak. Pro teoretickou část této bakalářské práce jsem se rozhodl čerpat z těchto zdrojů: [1], [2], [3], [4] a [5]. Každou část kapitoly teoretické části jsem inspiroval podle těchto zdrojů, proto jsem vyhodnotil, že je zbytečné tyto zdroje pod každou kapitolou citovat. Dále je tedy uvádět nebudu a budu uvádět pouze další zdroje, které budou spojené s konkrétní kapitolou.

Kapitola 2

Konečné automaty

Konečný automat (KA) je výpočetní model primitivního počítače, který se skládá z řídící jednotky a vstupní pásky. Řídící jednotka se může nacházet v některém z konečném počtu stavů, mezi kterými, na základě symbolu na pásce, řídící jednotka přechází. Kromě aktuálního stavu žádnou jinou paměť řídící jednotka nemá. Konečný automat je velice jednoduchý výpočetní model, dokáže rozpoznávat pouze *regulární jazyky*. Konečný automat je tedy ekvivalentní s regulárními výrazy a regulárními gramatikami. V této kapitole si podrobněji popíšeme základní druhy konečných automatů, se kterými budeme pracovat v dalším textu. Jedná se *deterministický konečný automat* (DKA) a *nedeterministický konečný automat* (NKA). Dále si popíšeme algoritmy, které s konečnými automaty pracují.

2.1 Deterministický konečný automat

Definice 1 *Deterministickým konečným automatem nazýváme pětice* $A = (Q, \Sigma, \delta, q_0, F)$*, kde:*

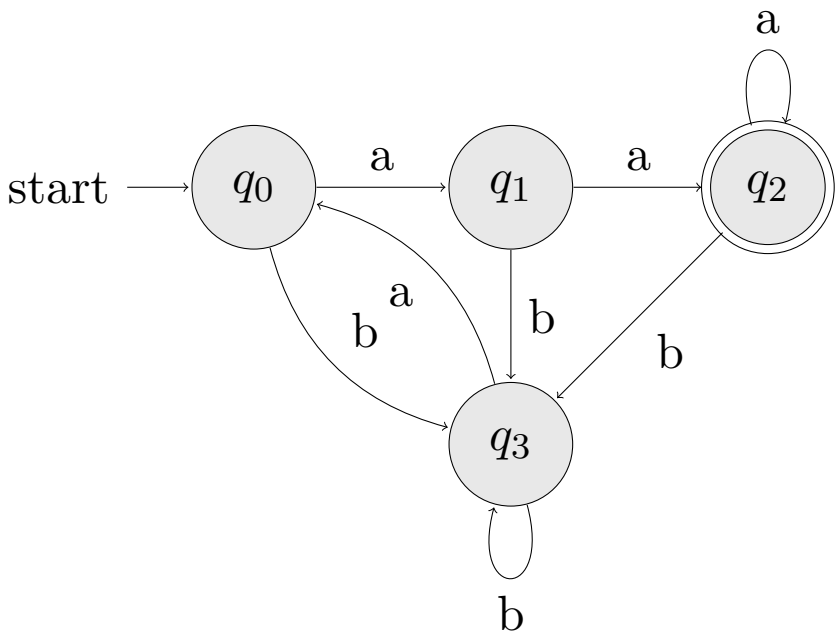
- Q *je konečná neprázdná množina stavů,*
- Σ *konečná neprázdná množina vstupních symbolů, nazývaná vstupní abeceda, či abeceda,*
- $\delta : Q \times \Sigma \to Q$ *je přechodová funkce,*
- $q_0 \in Q$ *je počáteční stav a*
- $F \subseteq Q$ *je množina přijímajících stavů.*

Poznámka 1 Přijímající stavy jsou někdy v české literatuře označovány jako koncové stavy.

2.1.1 Způsoby zadání deterministického konečného automatu

Deterministický konečný automat $A = (Q, \Sigma, \delta, q_0, F)$ lze zadat několika způsoby. Patrně nejběžnějším způsobem zadání DKA je grafické zadání *stavovým diagramem*, jehož hlavní výhodou je,

z lidského hlediska, především přehlednost zadání konečných automatů s malým počtem stavů[1]. Příklad takového grafického zadání můžeme vidět na obrázku 2.1. Stavový diagram má podobu orientovaného grafu, kde vrcholy grafu reprezentují jednotlivé stavy. Vrcholy grafu mají obvykle podobu kruhu s označením stavu uvnitř kruhu. Počáteční stav většinou označujeme krátkou šipkou ukazující na daný stav či stavy, přijímající stavy znázorňujeme dvojitým kruhem. DKA na obrázku 2.1 má tedy čtyři stavy, stav q_0 je počáteční stav, stav q_2 je přijímající stav, stavy q_1 a q_3 nejsou ani počáteční ani přijímající. Přechody mezi stavy jsou v diagramu znázorněny orientovanými hranami, kdy hrana spojující dva stavy je navíc ohodnocena symbolem abecedy, který je v průběhu přechodu KA z jednoho stavu do druhého přijímán. Například ze stavu q_0 vedou dvě hrany, přechody. Jedna hrana, ohodnocená symbolem a, vede do stavu q_1, druhá hrana vede do stavu q_3 a je ohodnocena symbolem b. Abeceda není ve stavovém diagramu explicitně zadána, kompletní abecedu lze získat sjednocením ohodnocení všech hran ve stavovém diagramu.



Obrázek 2.1: Stavový diagram DKA

Dále je možné zadat konečný automat pomocí tabulky, viz tabulka 2.1. Tabulka je v podstatě zadáním příslušné přechodové funkce δ. Ale lze z ní odvodit všechny potřebné údaje o daném konečném automatu. Množina stavů je definována v prvním sloupci tabulky. Šipka směřující doprava označuje počáteční stav, šipka směřující doleva označuje přijímací stavy, oboustranná šipky by pak definovala stav, který je současně počáteční i přijímací. V záhlaví sloupců jsou uvedeny postupně všechny symboly abecedy Σ.

[1]Stavový diagram můžeme také využít pro vizualizaci konečného automatu zadaného jiným způsobem.

δ	a	b
$\rightarrow q_0$	q_1	q_3
q_1	q_2	q_3
$\leftarrow q_2$	q_2	q_3
q_3	q_0	q_3

Tabulka 2.1: DKA zadaný tabulkou

Konečný automat je pochopitelně možné zadat výčtem všech množin a zobrazení uvedených v definici 1. Tento způsob zadání není z lidského hlediska příliš čitelný a intuitivní, ale je naopak velice výhodný pro počítačové zpracování. V tom případě můžeme místo matematického zápisu množin a zobrazení využít některého ze strojově čitelných formátů, například XML. Konečný automat $A = (Q, \Sigma, \delta, q_0, F)$ zadaný stavovým diagramem na obrázku 2.1 můžeme výčtem zadat takto:

- množina stavů $Q = \{q_0, q_1, q_2, q_3\}$,
- abeceda $\Sigma = \{a, b\}$,
- přechodová funkce

$$\begin{aligned}
\delta(q_0, a) &= q_1 \\
\delta(q_0, b) &= q_3 \\
\delta(q_1, a) &= q_2 \\
\delta(q_1, b) &= q_3 \\
\delta(q_2, a) &= q_2 \\
\delta(q_2, b) &= q_3 \\
\delta(q_3, a) &= q_0 \\
\delta(q_3, b) &= q_3
\end{aligned}$$

- počáteční stav q_0 a
- množina přijímajících stavů $F = \{q_2\}$.

2.1.2 Výpočet deterministického konečného automatu

Výpočet deterministického konečného automatu zahajujeme v počátečním stavu, který označíme za aktuální stav. Na základě symbolu na vstupní pásce a aktuálního stavu, řídící jednotka přejde, pomocí přechodové funkce, do nového aktuálního stavu. Pokud po zpracování všech symbolů na vstupní pásce dospěl konečný automat do jednoho z přijímajících stavů, je výpočet ukončen úspěšně. V opačném případě výpočet končí neúspěchem.

Výpočet DKA můžeme popsat pomocí *konfigurace DKA*. Konfigurace DKA je tvořena aktuální stavem a dosud nezpracovaným obsahem vstupní pásky, jinak řečeno dosud nezpracovanou částí vstupního slova. Formálně můžeme konfiguraci psát jako dvojici $Q \times \Sigma^*$. Například pro konečný automat na obrázku 2.1 je dvojice $(q_1, baaa)$ jednou z možných konfigurací. Na množině všech konfigurací konečného automatu můžeme definovat binární relaci symbolem $\vdash$ takovou, že $(q, w) \vdash (q', w')$ právě když $w = aw'$ a zároveň $q' = \delta(q, a)$. Pro dvě konfigurace C_1 a C_2 relace $C_1 \vdash C_2$, že automat může přejít jedním přechodem z konfigurace C_1 do konfigurace C_2. Konfigurace (q, w) se nazývá *počáteční konfigurace*, jestliže platí $q = q_0$. Konfigurace (q, w) se nazývá *koncová konfigurace*, jestliže platí $w = \varepsilon$. Koncová konfigurace (q, ε) se nazývá *přijímající*, jestliže $q \in F$. Automat *přijímá* slovo $w \in \Sigma^*$, právě tehdy, jestliže výpočet začínající v počáteční konfiguraci (q, w) skončí v přijímající koncové konfiguraci.

Příklad 1
Předpokládejme, že máme dán DKA z obrázku 2.1. Provedeme výpočet pro vstupní slovo *abaaa*. Výpočet zahájíme v počáteční konfiguraci $(q_0, abaaa)$. Během výpočtu DKA projde následující posloupností konfigurací

$$(q_0, abaaa) \vdash (q_1, baaa) \vdash (q_3, aaa) \vdash (q_0, aa) \vdash (q_1, a) \vdash (q_2, \varepsilon).$$

Koncová konfigurace (q_2, ε) je přijímající konfigurací, výpočet byl tedy úspěšný. ■

2.2 Nedeterministický konečný automat

Zobecněním deterministického konečného automatu dostáváme *nedeterministický konečný automat*, který je definován následujícím způsobem.

Definice 2 *Nedeterministickým konečným automatem nazýváme pětici $A = (Q, \Sigma, \delta, I, F)$, kde:*

- *Q je konečná neprázdná množina stavů,*
- *Σ je konečná neprázdná množina vstupních symbolů, nazývaná abeceda,*
- *$\delta : Q \times \Sigma \to P(Q)$ je přechodová funkce, symbolem $P(Q)$ označujeme potenční množinu množiny Q,*
- *$I \subseteq Q$ je množina počátečních stavů,*
- *$F \subseteq Q$ je množina přijímajících stavů.*

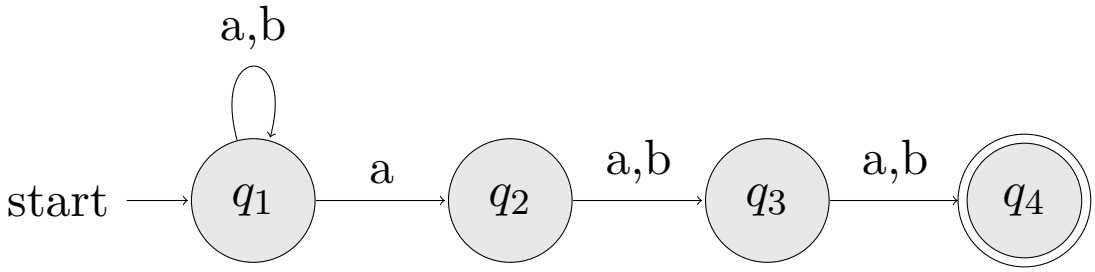
Srovnáme-li definice deterministického, Definice 1, a nedeterministického konečného automatu vidíme, že kromě více možných počátečních stavů, hlavní rozdíl spočívá v tom, že přechodová funkce

δ	a	b
$\rightarrow q_1$	q_1, q_2	q_1
q_2	q_3	q_3
q_3	q_4	q_4
$\leftarrow q_4$	-	-

Tabulka 2.2: NKA zadaný tabulkou

umožňuje přechod do více stavů současně. To znamená, že aktuální stav NKA v průběhu výpočtu není jediný stav, ale jedná se o celou množinu stavů. NKA se tak ocitá ve více stavech paralelně.

Příklad nedeterministického konečného automatu zadaného stavovým diagramem můžeme vidět na obrázku 2.2. Zadání stejného NKA pomocí tabulky je uvedeno v tabulce 2.2, kde je jasně vidět, že se jedná o NKA – přechod ze stavu q_1 symbolem a směřuje do dvou stavů q_1 a q_2 současně.



Obrázek 2.2: Stavový diagram NKA

2.2.1 Výpočet nedeterministického konečného automatu

Výpočet NKA je přirozeným rozšířením výpočtu DKA. Konfigurace NKA je opět určena aktuálním stavem řídící jednotky a dosud nezpracovaným obsahem vstupní pásky. Vzhledem k tomu, že z jednoho stavu lze přechodovou funkcí přejít stejným symbolem do více dalších stavů, aktuální stav řídící jednotky je určen pomocí množiny stavů, jde tedy o podmnožinu $P(Q)$, formálně je konfigurace NKA prvkem kartézského součinu $P(Q) \times \Sigma^*$. Například pro konečný automat na obrázku 2.2 je dvojice $(\{q_1, q_2\}, ab)$ jednou z možných konfigurací. Na množině všech konfigurací NKA můžeme definovat binární relaci symbolem $\vdash$ takovou, že $(X, w) \vdash (X', w')$ právě když $w = aw'$ a zároveň $X' = \bigcup_{\forall q \in X} \delta(q, a)$. Obdobným způsobem je nutné rozšířit další pojmy. Konfigurace (X, w) se nazývá *počáteční konfigurace*, jestliže platí $X = I$. Konfigurace (X, w) se nazývá *koncová konfigurace*, jestliže platí $w = \varepsilon$. Koncová konfigurace (X, ε) se nazývá *přijímající*, jestliže $X \cap F \neq \emptyset$, jinak řečeno v množině X existuje aspoň jeden přijímající stav. Automat *přijímá* slovo $w \in \Sigma^*$, právě tehdy, jestliže výpočet začínající v počáteční konfiguraci (q, w) skončí v přijímající koncové konfiguraci.

Příklad 2
Předpokládejme, že máme dán NKA z obrázku 2.2. Provedeme výpočet pro vstupní slovo *aaaa*. Výpočet zahájíme v počáteční konfiguraci $(\{q_1\}, aaaa)$. Během výpočtu NKA projde následující posloupností konfigurací

$$(\{q_1\}, aaaa) \vdash (\{q_1, q_2\}, aaa) \vdash (\{q_1, q_2, q_3\}, aa) \vdash (\{q_1, q_2, q_3, q_4\}, a) \vdash (\{q_1, q_2, q_3, q_4\}, \varepsilon).$$

Koncová konfigurace $(\{q_1, q_2, q_3, q_4\}, \varepsilon)$ je přijímající konfigurací, protože $\{q_1, q_2, q_3, q_4\} \cap F = \{q_4\}$. Výpočet byl tedy úspěšný. ■

2.3 Zobecněný nedeterministický konečný automat

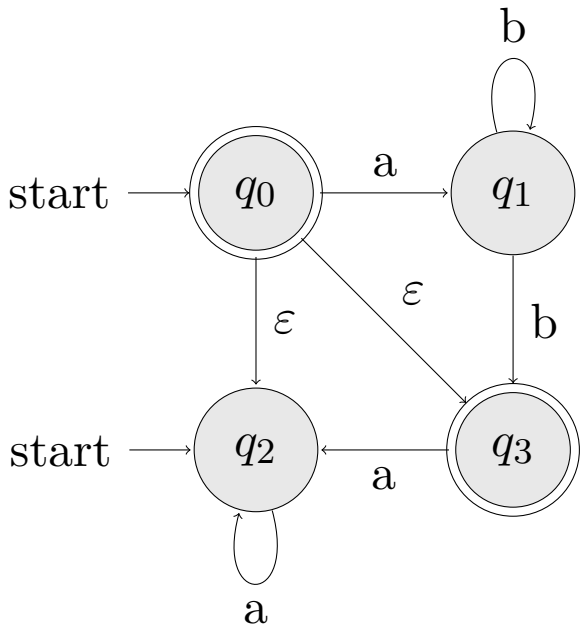
Zobecněný nedeterministický konečný automat (ZNKA) je dalším rozšířením NKA. ZNKA může obsahovat přechody mezi stavy automatu při kterých nedochází k přijetí znaku ze vstupní pásky. Tyto přechody se nazývají ε-přechody – ε-přechod můžeme chápat i tak, že při provedení tohoto přechodu se ze vstupní pásky zpracuje prázdný znak. Pokud při výpočtu se ZNKA dostane do stavu, který obsahuje jeden nebo více ε-přechodů, posune se ihned do stavu či stavů, kam ε-přechody ukazují. To se opakuje do doby, dokud nenarazí na stav nebo stavy, které neobsahují ani jeden ε-přechod a pak pokračuje dále v přijímání symbolů ze vstupní pásky. Pokud se na pásce nenachází žádný symbol a automat je ve stavu, který obsahuje ε-přechody, řídící jednotka se posune i do těchto stavů, včetně stavu původního.

Poznámka 2 Může nastat situace, kdy se automat během výpočtu dostane i do nekonečného cyklu. Toto nebezpečí hrozí v případě že stavový diagram obsahuje orientovaný cyklus tvořený ε-přechody.

Definice 3 *Zobecněným nedeterministickým konečným automatem nazýváme pětici* $A = (Q, \Sigma, \delta, I, F)$, *kde:*

- Q *je konečná neprázdná množina stavů,*
- Σ *je konečná abeceda,*
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to P(Q)$ *je zobecněná přechodová funkce,*
- $I \subseteq Q$ *je množina počátečních stavů a*
- $F \subseteq Q$ *je množina přijímajících stavů.*

Příklad stavového diagramu zobecněného nedeterministického konečného automatu můžeme vidět na obrázku 2.3.



Obrázek 2.3: Příklad ZNKA

2.4 Redukce konečného automatu

V této kapitole si povíme, jakým způsobem se automaty redukují, aniž bychom změnili jejich ekvivalentnost.

Redukce KA si zakládá na odstranění přechodů a stavů, přesněji nadbytečných, nedosažitelných a ekvivalentních stavů a přechodů mezi nimi. Pro každý případ je vyčleněna podkapitola.

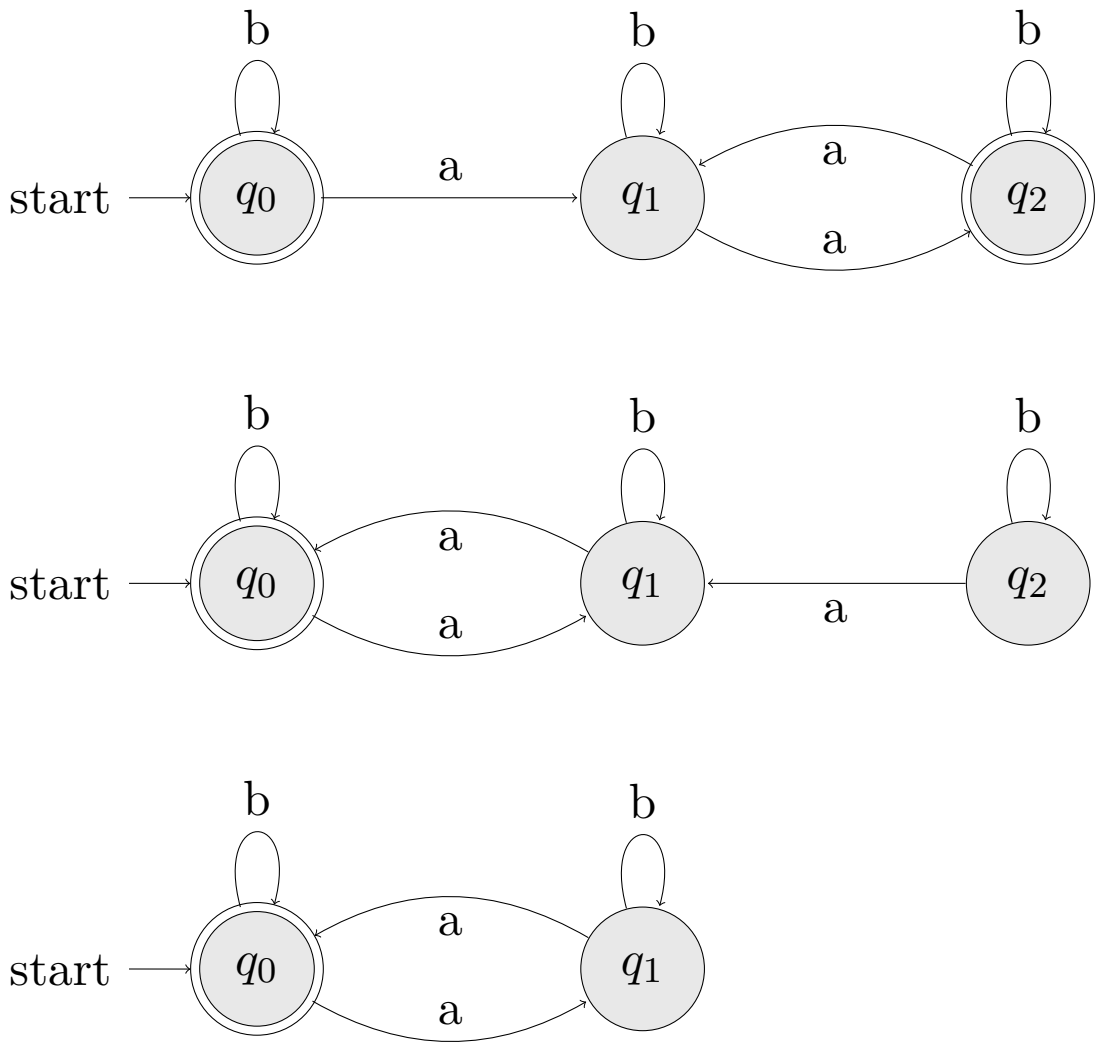
Poznámka 3 V některých literaturách se redukce automatu také označuje jako minimalizace.

Definice 4 *Automat nazýváme redukovaný, jestliže neexistuje ekvivalentní automat s menším počtem stavů.*

2.4.1 Ekvivalence automatů

Pokud máme automaty, které se liší počtem stavů, nebo přechodů, ale přijímají stejný jazyk, nazýváme je ekvivalentní.

Definice 5 *O konečných automatech* $A1$ *a* $A2$ *řekneme, že jsou ekvivalentní, jestliže* $L(A1) = L(A2)$.



Obrázek 2.4: Příklady konečných automatů

Poznámka 4 Na obrázku 2.4 jsou tři příklady automatů. Všechny tyto automaty přijímají jazyk všech slov se sudým počtem symbolu a. Automat uprostřed zároveň obsahuje nedosažitelný stav q_2.

2.4.2 Nedosažitelné stavy automatů

Pokud existují stavy, do kterých se při výpočtu řídící jednotka nedostane ani pro jednu posloupnost vstupních symbolů, nazýváme tyto stavy nedosažitelné.

Nedosažitelné stavy, a všechny přechody vedoucí do nich a z nich, můžeme z automatu odstranit. Jazyk přijímaný automatem se nezmění.

Abychom odstranili nedosažitelné stavy musíme si vytvořit množinu E a vložit do ní všechny počáteční stavy ($E = I$). Tuto množinu rozšíříme o všechny stavy, do kterých vedou přechody

z původní množiny E. Tento cyklus opakujeme do doby, dokud není možné do množiny E přidat další stav.

Definice 6 *Stav q konečného automatu $A = (Q, \Sigma, \delta, q_0, F)$ není dosažitelný, pokud neexistuje nějaké slovo w takové, že $q_0 \overrightarrow{w} q$.*

2.4.3 Nadbytečné stavy

Pokud v automatu existují stavy, ze kterých se pro žádnou posloupnost symbolů ze vstupní pásky není možné dostat do stavů přijímajících, jsou tyto stavy nadbytečné. Výpočet automatu zpomalují, a proto je můžeme z automatu odstranit. Automat bude i po odstranění nadbytečných stavů sám sobě ekvivalentním.

Abychom odstranili nadbytečné stavy, musíme nejdříve odstranit stavy nedosažitelné. Poté si vytvoříme množinu E, která bude obsahovat všechny přijímající stavy ($E = F$). Nyní tuto množinu rozšíříme o všechny stavy, ze kterých vede přechod do stavů, které se už v množině E nachází. Tuto činnost opakujeme do doby, dokud nebude možné do množiny E přidat další stav.

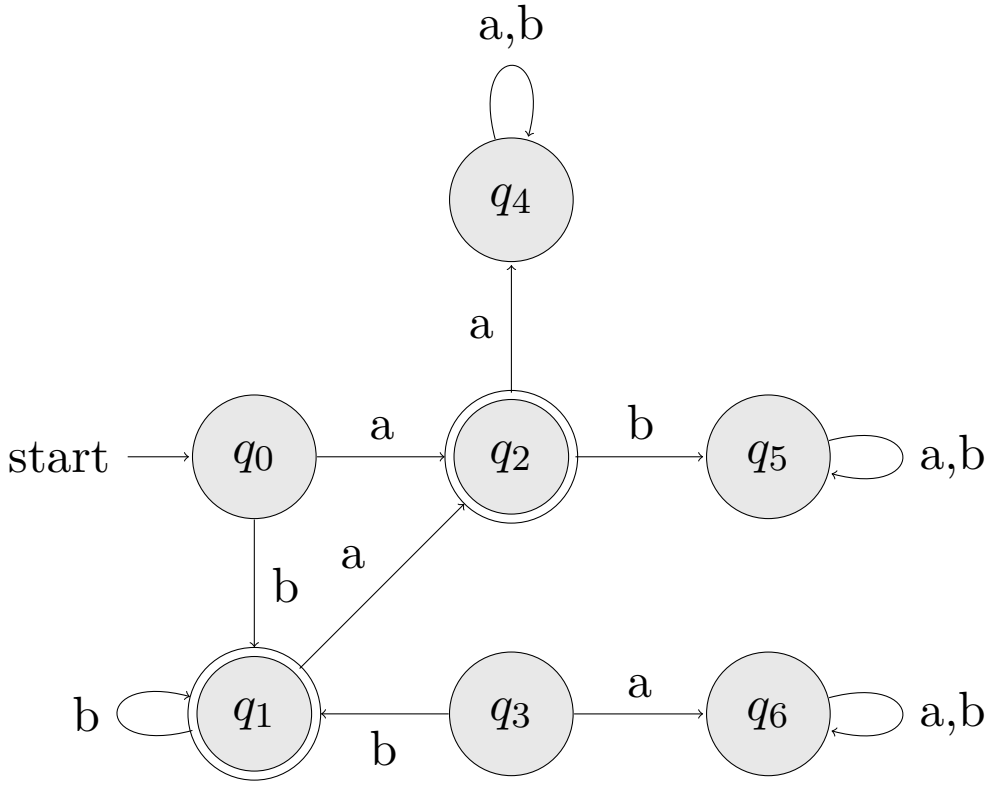
Definice 7 *Stav $q \in Q$ je nadbytečný, pokud neexistuje žádné slovo $w \in \Sigma^+$ a žádný stav $q_f \in F$ takový, že $\langle q, w \rangle \rightarrow^* \langle q_f, \varepsilon \rangle$*

2.4.4 Ekvivalentní stavy

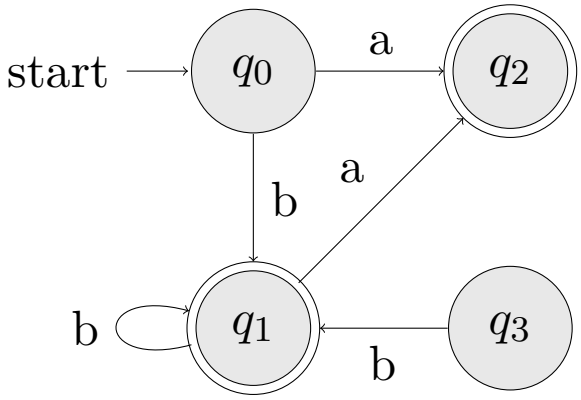
Dalším typem stavů, které můžeme z automatu odstranit, a přitom nezměnit ekvivalentnost automatu, jsou stavy ekvivalentní. Tyto stavy nemusí být ani nedosažitelné, ani nadbytečné, nicméně pokud jsou nějaké dva nebo více stavů ekvivalentní, znamená to, že v automatu zastávají stejný význam a víc než jeden takových stavů automat pro správný výpočet nepotřebuje.

Příklad 3
Automat na obrázku 2.7 si zvolíme jako příklad. Nejdříve si musíme všechny stavy rozdělit do dvou množin. Do první množiny stavů zahrneme všechny stavy, které nejsou přijímající, druhá množina bude obsahovat všechny stavy přijímající. Znamená to tedy, že množina $M_1 = \{q_0, q_1, q_2, q_4, q_5, q_6\}$ a množina $M_2 = \{q_3, q_7\}$.

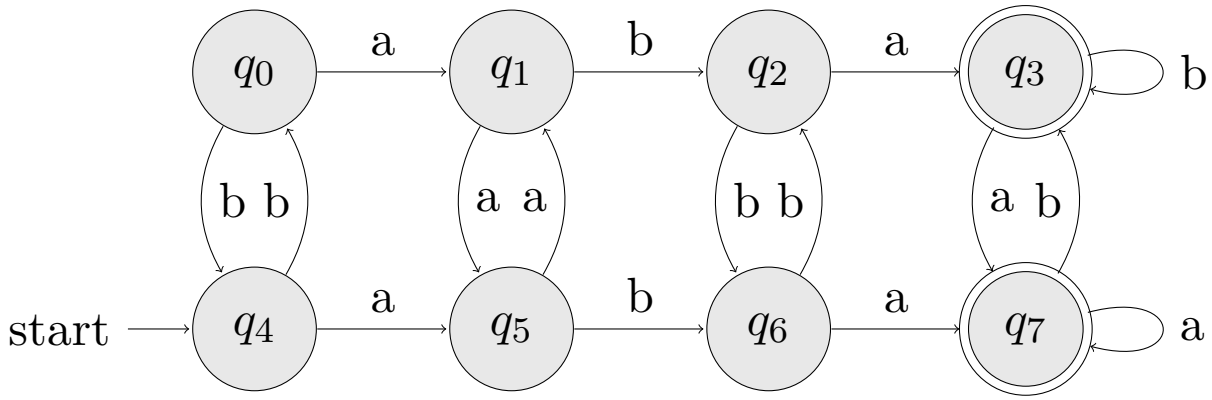
Pro odstranění ekvivalentních stavů použijeme tabulku. Ta obsahuje sloupec pro množiny stavů, sloupec pro stavy samotné, a poté jeden sloupec pro každý unikátní symbol z abecedy.



Obrázek 2.5: Automat před odstraněním nadbytečných stavů



Obrázek 2.6: Automat po odstranění nadbytečných stavů



Obrázek 2.7: DKA před odstraněním ekvivalentních stavů

Množina	Stav	a b
M_1	q_0 q_1 q_2 q_4 q_5 q_6	
M_2	q_3 q_7	

Tabulka 2.3: Výchozí tabulka algoritmu před prvním zápisem

Nyní do tabulky budeme zapisovat. Nebudeme však zapisovat stavy, do kterých se řídící jednotka dostane po přechodových funkcích, ale budeme zapisovat množiny, do kterých tyto stavy patří. Např. stav q_0 se po přechodu a ocitne ve stavu q_1. My však do tabulky nebudeme zapisovat stav q_1, nýbrž množinu, do které stav q_1 patří, takže M_1.

Po zápisu vypadá tabulka takto:

Množina	Stav	a	b
M_1	q_0	M_1	M_1
	q_1	M_1	M_1
	q_2	M_2	M_1
	q_4	M_1	M_1
	q_5	M_1	M_1
	q_6	M_2	M_1
M_2	q_3	M_2	M_2
	q_7	M_2	M_2

Tabulka 2.4: Tabulka po prvním kroku algoritmu

Všimněme si, že stavy q_2 a q_6 se odlišují od zbytku stavů. Tyto stavy vyjmeme, vytvoříme novou množinu a vložíme do ní vyjmuté stavy z M_1. Stavy q_0, q_1, q_4 a q_5 ponecháme v množině M_1, protože se od sebe neodlišují.

Protože jsme vytvořili novou množinu, tak zapsaná tabulka již není validní, musíme ji upravit.

Množina	Stav	a	b
M_1	q_0	M_1	M_1
	q_1	M_1	M_2
	q_4	M_1	M_1
	q_5	M_1	M_2
M_2	q_2	M_3	M_2
	q_6	M_3	M_2
M_3	q_3	M_3	M_3
	q_7	M_3	M_3

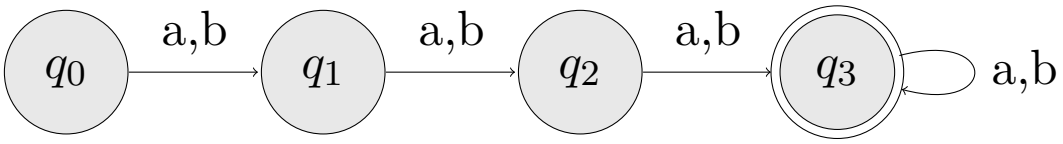
Tabulka 2.5: Tabulka po druhém kroku algoritmu

Opět můžeme vidět, že v množině M_1 jsou stavy, které se od sebe liší koncovou množinou. Obdobně jako v minulém kroku vyčleníme pro tyto stavy novou množinu a opět tabulku přepíšeme do korektního stavu.

Množina	Stav	a	b
M_1	q_0	M_2	M_1
	q_4	M_2	M_1
M_2	q_1	M_2	M_3
	q_5	M_2	M_3
M_3	q_2	M_4	M_3
	q_6	M_4	M_3
M_4	q_3	M_4	M_4
	q_7	M_4	M_4

Tabulka 2.6: Tabulka po třetím kroku algoritmu

Nyní už nepotřebujeme provádět další kroky algoritmu. Ve všech množinách jsou stavy k sobě ekvivalentní. Do výsledného automatu zahrneme pouze jeden stav z každé množiny. Počáteční stav bude z té samé množiny, která obsahuje původní počáteční stav a přijímající stav bude z množiny přijímajících stavů.



Obrázek 2.8: Automat po odstranění ekvivalentních stavů

■

Definice 8 *Stavy q_1 a q_2 nazýváme ekvivalentní, jestliže q_{x_1}, q_{x_2} až q_{x_n} mají všechny přechodové funkce shodné s přechodovými funkcemi pro $q_{x_1} \times \Sigma \to q_m$.*

2.5 Převod zobecněného nedeterministického konečného automatu na deterministický konečný automat

V této části je popsán převod ZNKA na DKA.

Převod ze ZNKA na DKA není vždy výhodný. Pokud máme ZNKA o n stavech, tak ekvivalentní převedený DKA může v nejhorším případě obsahovat až 2^n stavů.

Převod vypadá takto:

1. Vytvoříme první stav, který se skládá z množiny I vstupních stavů.
2. Pokud některý z těchto stavů obsahuje ε-přechody, zahrneme do množiny počátečních stavů i stavy, do kterých se lze dostat pomocí těchto ε-přechodů.

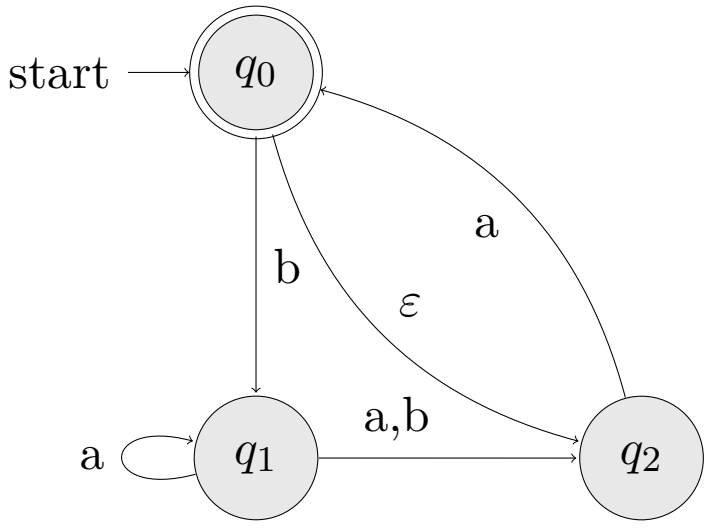
3. Přečteme první symbol ze vstupní pásky. Vytvoříme nový stav, do kterého bude vést přechod z počátečního stavu přes daný symbol. Pokud tento nový stav obsahuje alespoň jeden z původních přijímajících stavů z množiny F, označíme tento stav v DKA jako přijímající.

4. Takhle přečteme všechny symboly ze vstupní pásky a vytvoříme k nim ekvivalentní stavy a přechody z původního ZNKA.

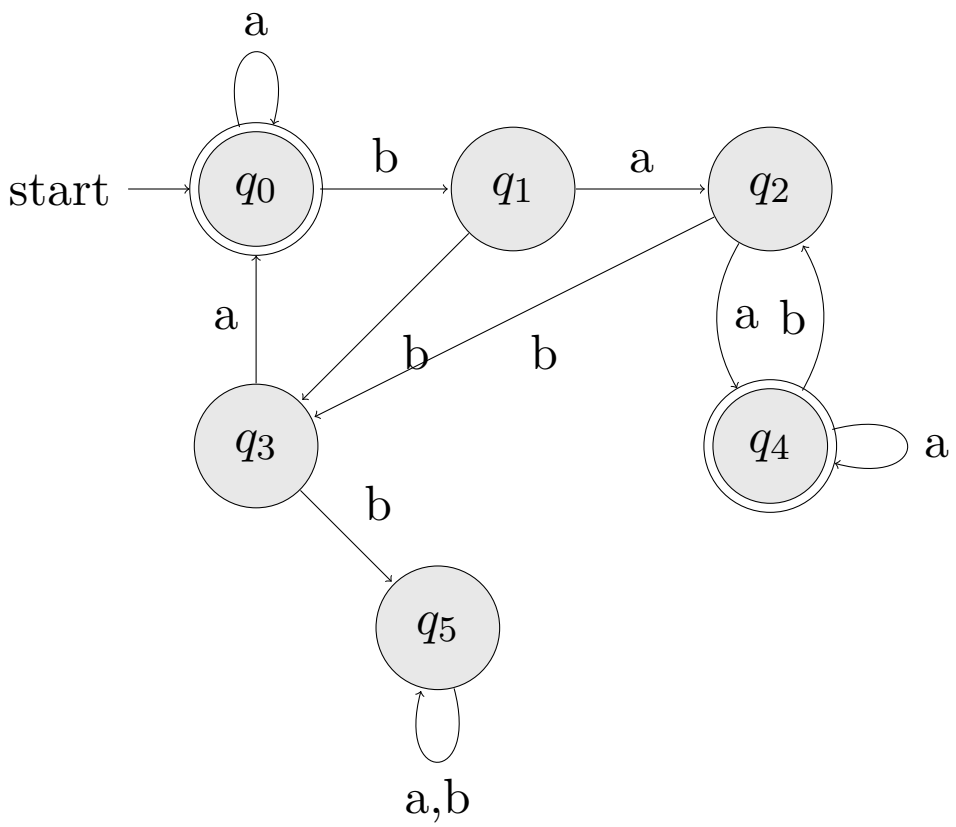
5. Pokud se nově vytvořený přijímající stav může dostat přes ε-přechod do jiného stavu, označíme tento stav také jako přijímající.

6. Pokud se řídící jednotka nachází v množině stavů, ke které neexistuje nadefinovaná přechodová funkce přes nějaký symbol z abecedy Σ, vytvoříme nový nepřijímající stav, označíme jej symbolem prázdné množiny ($\emptyset$), a povedeme do něj přechod daným znakem. Tento nový stav zacyklíme pro všechny znaky z abecedy Σ.

Poznámka 5 Řídící jednotka v ZNKA může nabývat současně více stavů a je pravděpodobné, že se to promítne i při převodu na DKA. Například algoritmus při převádění se může nacházet ve stavu q_3, který má definovanou přechodovou funkci symbolem x do stavů q_1, q_4 a q_5. V tomto případě vytvoří algoritmus pouze jeden stav, který bude zastávat pozici těchto tří stavů a k tomuto stavu vytvoří přechodovou funkci přes symbol x.



Obrázek 2.9: Automat před převodem na DKA



Obrázek 2.10: Automat po převodu na DKA

Poznámka 6 Nově vytvořené stavy v DKA si můžeme přejmenovat podle vlastního uvážení. Na obrázku 2.10 můžeme vidět převedený automat z původního ZNKA. V nově vytvořeném DKA byl stav q_4 původně nazván stavem q_0, q_1, q_2 a např. stav q_5 byl nazván jako $\varnothing$.

Definice 9 *K zobecněnému nedeterministickému konečnému automatu* $A = (Q, \Sigma, \delta, I, F)$ *můžeme nyní sestrojit deterministický konečný automat* $A' = (Q', \Sigma, \delta', q'_0, F')$*, kde :*

- $Q' = P(Q)$
- $\delta' : Q' \times \Sigma \to Q'$
- $q'_0 = Cl_\varepsilon(I)$
- $F' = \{K \in Q' | Cl_\varepsilon(K) \cap F \neq \varnothing\}$

Dále platí $L(A1) = L(A2)$

2.6 Odstranění epsilon přechodů

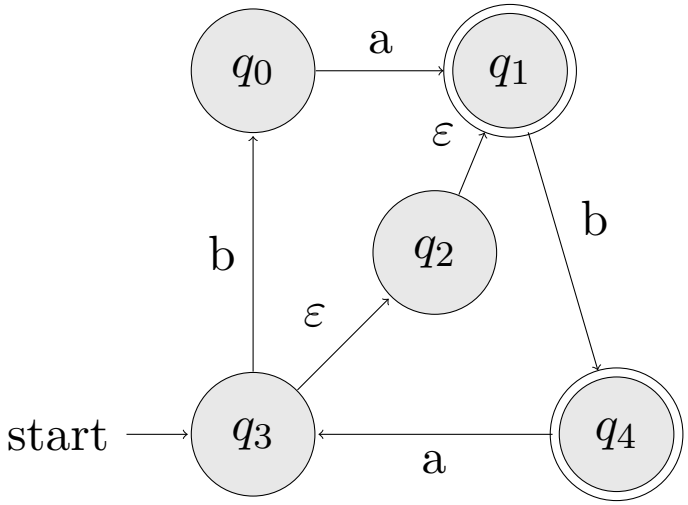
Ke každému ZNKA existuje alespoň jeden ekvivalentní NKA. Ten dostaneme tak, že z původního ZNKA odstraníme ε-přechody pomocí následujícího algoritmu:

1. Ke každému stavu vytvoříme epsilon uzávěr.

2. Smažeme všechny ε-přechody.

3. Pro každou hranu vedoucí do stavu S uděláme kopii vedoucí do všech stavů z ε-uzávěru původního stavu S.

4. Počáteční budou všechny stavy, které jsou v uzávěru alespoň jednoho počátečního stavu.



Obrázek 2.11: Vybraný automat k odstranění ε-přechodů

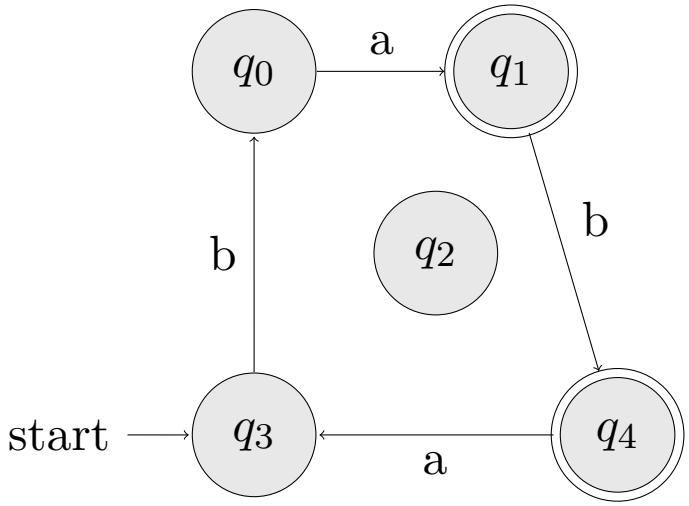
Příklad 4
Na obrázku 2.11 je vyobrazen automat, kterému odstraníme ε-přechody.

Epsilon uzávěry můžeme zapisovat do tabulky. Tabulka má dva sloupce. První sloupec reprezentuje všechny stavy v automatu. Druhý sloupec už je pro samotné ε-uzávěry. Mějme například stav q_3 na obrázku 2.11, který se pomocí ε-přechodů může dostat do stavů q_2 i q_1, proto do ε-uzávěru napíšeme i tyto stavy. Do ε-uzávěru píšeme i samotný výchozí stav, v tomto případě i stav q_3.

Stav	ε-uzávěr
q_0	q_0
q_1	q_1
q_2	q_2,q_1
q_3	q_3,q_2,q_1
q_4	q_4

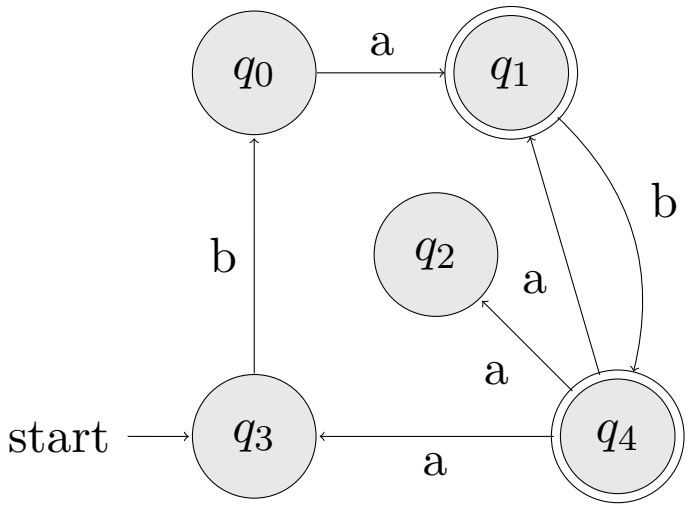
Tabulka 2.7: Tabulka ε-uzávěrů všech stavů

Dále smažeme všechny ε-přechody.



Obrázek 2.12: Vybraný automat po odstranění ε-přechodů

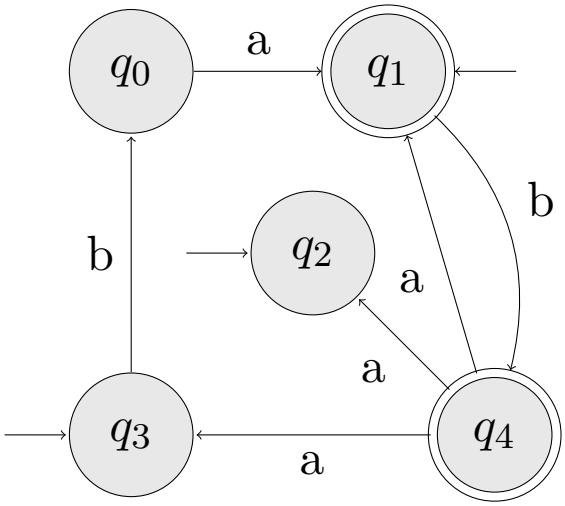
Nyní vytvoříme kopie každé hrany, která vede do nějakého stavu S, který má více než jeden stav ve svém ε-uzávěru. Tyto kopie budou vést do stavů z ε-uzávěru stavu S. Pokud má stav pouze jeden stav ve svém ε-uzávěru, znamená to, že má ve svém ε-uzávěru sám sebe a není nutné proto vytvářet kopie hran.



Obrázek 2.13: Vybraný automat po přidání nových přechodů

Přidáme 2 přechody ze stavu q_4 do stavů q_2 a q_1 přes symbol a.

Posledním krokem je označit všechny stavy, které se nachází v ε-uzávěru alespoň jednoho počátečního stavu, jako počáteční.



Obrázek 2.14: Vybraný automat po přidání nových počátečních stavů

■

Kapitola 3

Regulární výrazy

Regulární výraz je pravidlo, které popisuje celou množinu řetězců. Je podskupinou formálních gramatik. Obecně RV popisuje jazyk nad libovolnou abecedou. V dnešní době se nejčastěji používá například pro práci s textem, vytahování dat z textu. Regulární výrazy popisující jazyky nad abecedou Σ: $\emptyset, \varepsilon, a$ (kde $a \in \Sigma$) jsou regulární výrazy:

- $\emptyset$ označuje prázdný jazyk
- ε označuje jazyk ε
- a označuje jazyk a

Jestliže α, β jsou regulární výrazy, pak i $(\alpha|\beta)$, $(\alpha.\beta)$, (α^*), (α^+), $(\alpha^?)$ jsou regulární výrazy:

- $(\alpha|\beta)$ označuje sjednocení jazyků α a β
- $(\alpha.\beta)$ označuje zřetězení jazyků α a β
- (α^*) označuje iteraci jazyka α
- (α^+) označuje nenulovou iteraci jazyka α
- $(\alpha^?)$ označuje nulovou nebo právě jednu iteraci jazyka α

3.1 Převod regulárního výrazu na zobecněný nedeterministický konečný automat

Abychom mohli převod realizovat, musíme mít k dispozici **parser**, který přečte vstupní řetězec (regulární výraz) a na základně nadefinovaného chování rozdělí řetězec na části, ze kterých je možné sestavit ZNKA. Teoretický popis **parseru** a následná implementace je popsána v [6].

Využijeme **parser**, který bude dělit vstupní řetězec na elementární RV, sjednocení, zřetězení nebo iteraci jazyků pomocí rekurzivního sestupu. Taktéž si musíme nadefinovat formální gramatiku. **Parser**, který dělí vstupní řetězec na základě rekurzivního sestupu má nadefinované funkce pro každý neterminál. **Parser** k práci využívá LL(1) gramatiku, což znamená, že čte řetězec zleva doprava, konstruuje nejlevější derivaci a zná maximálně jeden následující symbol.

Definice formální gramatiky v BNF notaci:

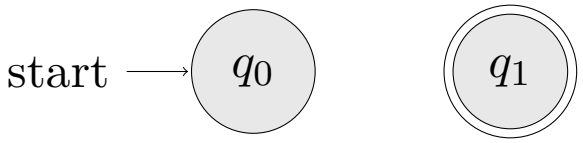
```
<expr> ::= <term>
    | <term>'|'<expr>
<term> ::= <factor>
    | <factor><term>
<factor> ::= <atom>
    | <atom><meta-char>
<atom> ::= <char>
    | '('<expr>')'
<char> ::= <any-char-except-meta>
    | '\'<any-char>
<meta-char> ::= '?' | '*' | '+'
```

Pro regulární výraz $a^+b^?$ vytvoří **parser** strom (anglicky *parse tree*), který je uveden na obrázku 3.1. Další příklad regulárního výrazu může být výraz $(a|b)^*c$ a jemu odpovídající strom na obrázku 3.2.

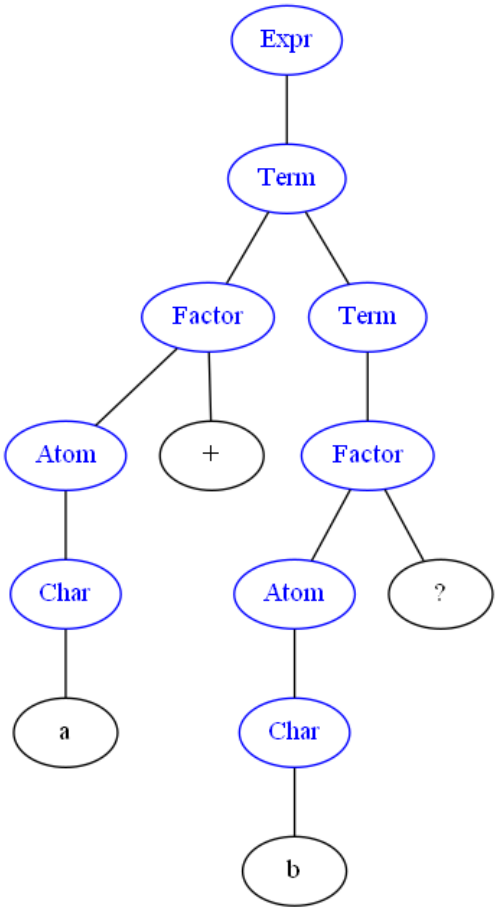
Pokud se například podíváme do definice formální gramatiky, tak neterminál $\langle atom \rangle$ je definován buď neterminálem *atom* nebo dalším výrazem (*expression*). Každý z dvou výše zmíněných RV nám demonstruje jiné využití z definice neterminálu *atom*. V prvním příkladu je *atom* pouze *char* písmeno a. Druhý příklad poukazuje, že *atom* je výraz: $(a|b)$.

Jakmile je RV rozdělený na části, může program začít sestavovat automat. K tomu jsou potřeba předdefinované funkce, které jsou schopné z jednotlivých částí vytvořit menší automaty a dále je spojit do sebe.

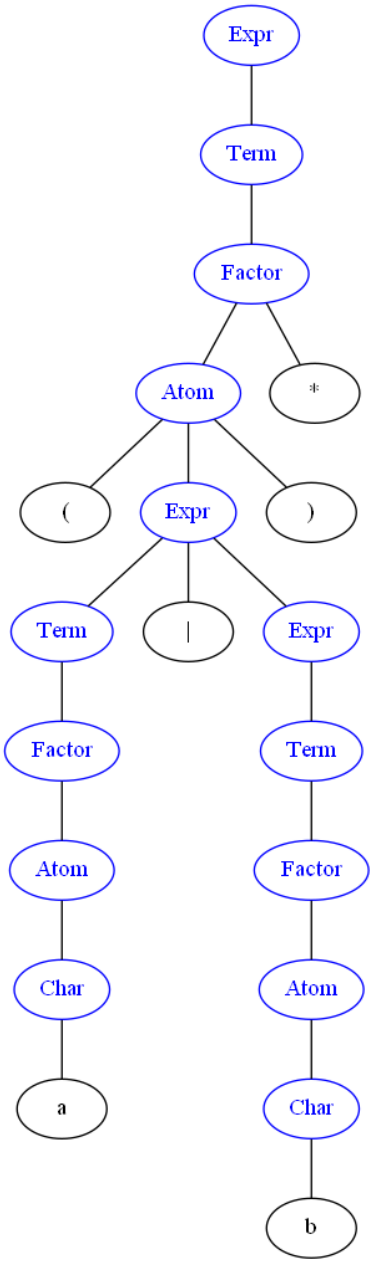
Zde jsou konstrukce pro elementární RV nebo pro sjednocení, zřetězení a iterace jazyka:



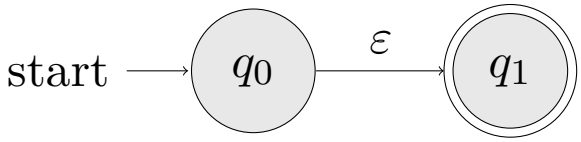
Obrázek 3.3: Automat pro prázdný jazyk



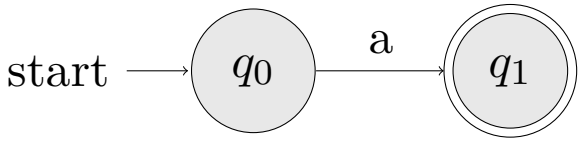
Obrázek 3.1: Parse tree prvního RV



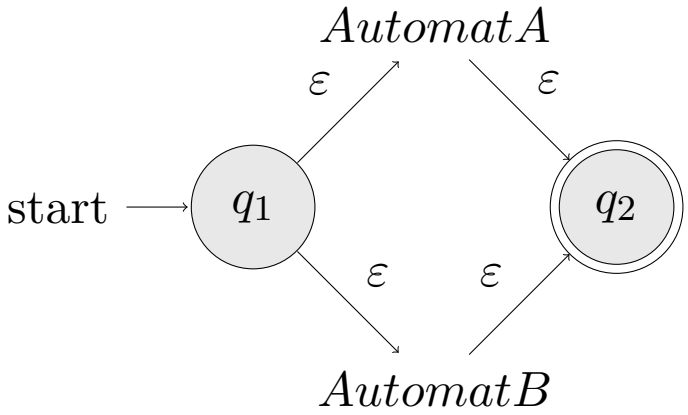
Obrázek 3.2: Parse tree druhého RV



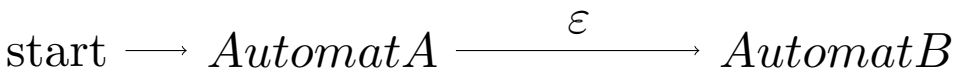
Obrázek 3.4: Automat pro ε přechod



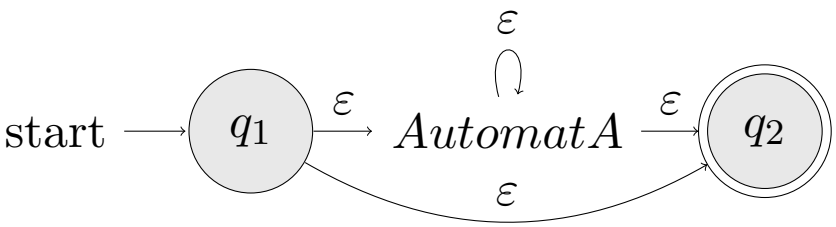
Obrázek 3.5: Automat pro přechod symbolem a



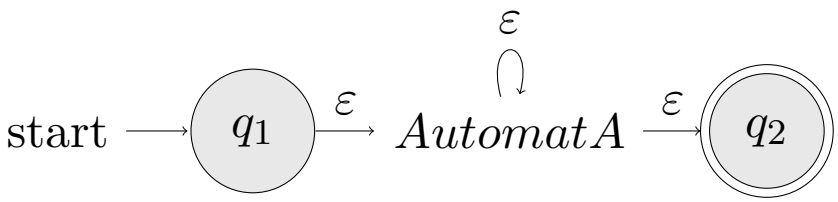
Obrázek 3.6: Konstrukce automatu pro sjednocení jazyků



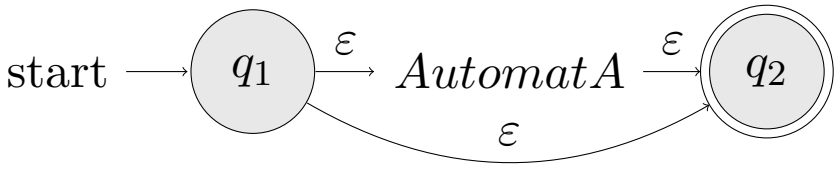
Obrázek 3.7: Konstrukce automatu pro zřetězení jazyků



Obrázek 3.8: Automat pro iteraci jazyka

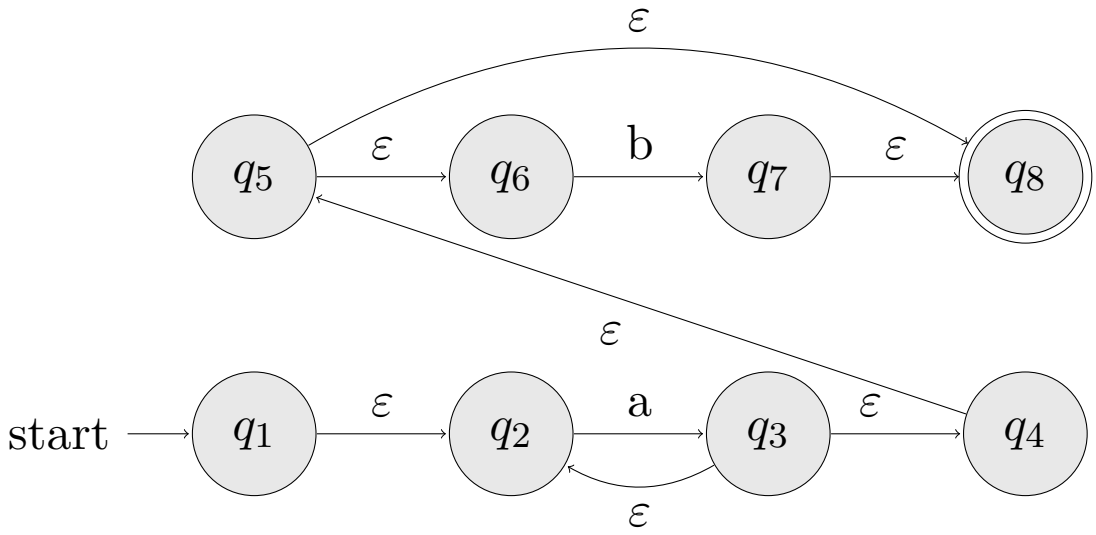


Obrázek 3.9: Automat pro nenulovou iteraci jazyka



Obrázek 3.10: Automat pro nulovou nebo právě jednu iteraci jazyka

Výsledný automat regulárního výrazu $a^+b^?$:



Obrázek 3.11: Výsledný automat pro RV $a^+b^?$

3.2 Derivace regulárního výrazu

Regulární výrazy můžeme derivovat podle libovolného symbolu z abecedy.

Definice 10 *Pro derivaci regulárního výrazu platí tato pravidla:*

- $\frac{d\varnothing}{da} = \varnothing, \forall a \in X$
- $\frac{d\varepsilon}{da} = \varnothing, \forall a \in X$

- $\frac{da}{da} = \varepsilon, \forall a \in X$
- $\frac{db}{da} = \varnothing, \forall b \neq a$
- $\frac{d(U+V)}{da} = \frac{dU}{da} + \frac{dV}{da}$
- $\frac{d(U.V)}{da} = \frac{dU}{da}.V, \varepsilon \notin U$
- $\frac{d(U.V)}{da} = \frac{dU}{da}.V + \frac{dV}{da}, \varepsilon \in U$
- $\frac{dV^*}{da} = \frac{dV}{da}.V^*$
- $\frac{dV}{dx} = \frac{d}{a_n}\left(\frac{d}{a_{n-1}}\left(\cdots \frac{dV}{a_1}\right)\right), x = a_1 \ldots a_{n-1}a_n$

3.2.1 Převod derivace regulárního výrazu na zobecněný nedeterministický konečný automat

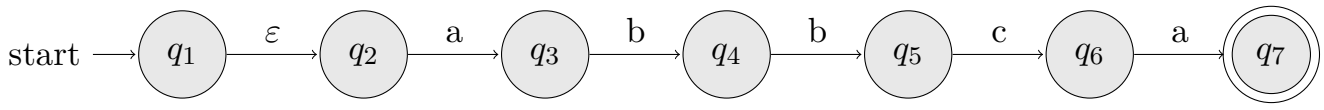
V této bakalářské práci využívám velmi zjednodušenou formu převodu, protože program umí zderivovat pouze množinu slov z jazyka $h(V)$ pro kterou platí

$$h\left(\frac{dV}{dx}\right) = \{v | xv \in h(V)\}.$$

V množině jsou všechna slova zkrácena o první znak a je k nim ekvivalentně vytvořen stav s přechodem. Algoritmus postupně zkracuje slova dále a vytváří k nim stavy s přechody, dokud nezpracuje všechna slova a všechny jejich znaky. Pokud množina obsahuje například čtyři řetězce, vytvoří se k nim ekvivalentní čtyři automaty, které mají společný počáteční stav.

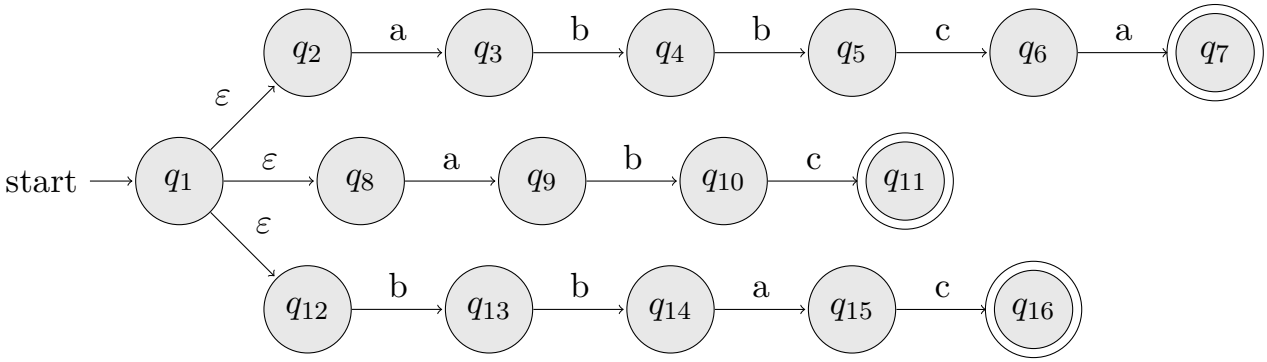
Poznámka 7 Tato bakalářská práce by se dala rozšířit, nebo by mohla být námětem na další bakalářskou či diplomovou práci. Rozšířit by se dala funkčnost algoritmu, aby mohl symbolicky zpracovávat derivace na základě definovaných pravidel v kapitole 3.2, a ne jenom se slovy nad abecedou.

Příklad 5
Mějme jako příklad množinu, která obsahuje jedno slovo: *abbca*. Pokud by program přijal na vstupu takovou množinu, výsledný automat by vypadal následovně:



Obrázek 3.12: První automat vytvořený z derivace RV

Kdybychom množinu rozšířili o další slova *abc* a *bbac*, tak by výsledný automat vypadal takto:



Obrázek 3.13: Druhý automat vytvořený z derivace RV

■

Kapitola 4

Regulární gramatika

Regulární gramatika je podskupinou formálních gramatik. Její pravidla jsou definována v těchto formách:

1. $A \rightarrow a$
2. $A \rightarrow aB$
3. $A \rightarrow \varepsilon$

A a B jsou neterminály, a je terminál a ε je znak pro prázdný řetězec.

Tato bakalářská práce pojednává o tzv. **right-regular grammar** , což znamená, že pokud jednotlivá pravidla obsahují na pravé straně rovnice neterminál, bude vždy co nejvíce vpravo.

4.1 Převod regulární gramatiky na zobecněný nedeterministický konečný automat

Program přijímá seznam pravidel. Jednotlivá pravidla se zapisují v BNF notaci.

```
List<string> RegularGrammar = new List<string>();

RegularGrammar.Add("<S> ::= 'abc'<A> | 'b'<B>");
RegularGrammar.Add("<A> ::= 'bbaa'<B> | 'ε'");
RegularGrammar.Add("<B> ::= 'a'<A> | 'bb'");
```

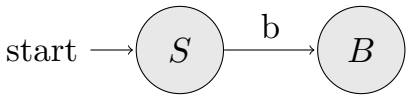
Listing 4.1: Vytváření pravidel pro regulární gramatiku

Do ostrých závorek zapisujeme neterminály. Do apostrofů píšeme terminály.

Algoritmus ze vstupního seznamu pravidel gramatiky nejdříve sestaví abecedu ze všech terminálů a vytvoří stavy ze všech neterminálů, a poté vytvoří ještě jeden konečný stav. Počátečním stavem bude stav nesoucí pojmenování po neterminálu z prvního pravidla, z ukázkového zdrojového kódu to je stav S. Každé pravidlo má dvě strany, levou a pravou. Levá strana označuje výchozí stav a pravá strana naznačuje po jakých přechodech se z výchozího stavu dostaneme do dalšího stavu.

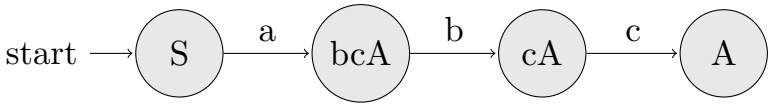
Příklad 6

Pokud se podíváme na první pravidlo příkladu v úvodu kapitoly 4.1, tak neterminál na levé straně $\langle S \rangle$ nám značí výchozí stav a jeho část pravé strany $'b'\langle B \rangle$ znamená, že se řídící jednotka po přechodu b dostane do stavu B.



Obrázek 4.1: Automat druhé části pravé strany prvního pravidla

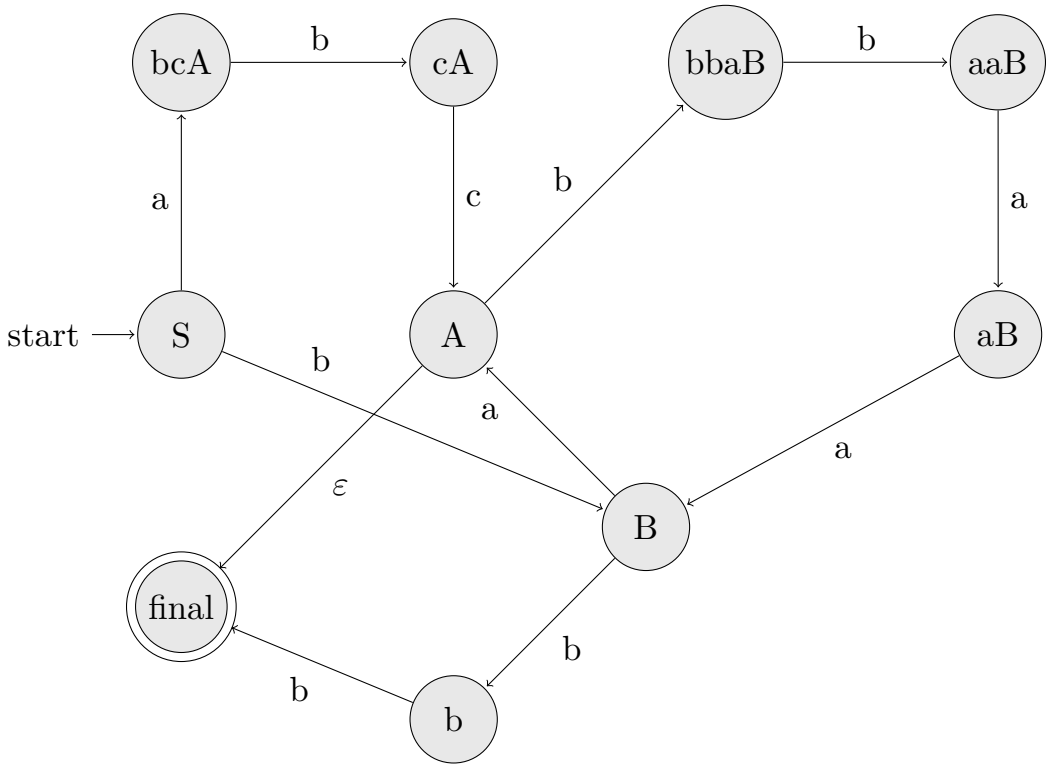
Pravá strana pravidla může mít více částí, které se oddělují symbolem |. Jednotlivé části pravidel můžou mít více terminálů. $\langle S \rangle ::='abc'\langle A \rangle$ nám značí, že ze stavu S se do stavu A dostaneme až po přechodech a, b a c. V praxi se taková situace řeší vytvořením dalších stavů bcA a cA, a poté přidáme přechody ze stavu S do stavu bcA přes symbol a, ze stavu bcA do stavu cA po přechodu b a konečně ze stavu cA do stavu A přes symbol c.



Obrázek 4.2: Automat první části pravé strany prvního pravidla

Výsledný automat původního příkladu na začátku kapitoly 4.1 vypadá takto:

■



Obrázek 4.3: Výsledný automat definovaný regulární gramatikou

Kapitola 5

Implementace

Všechny funkce a algoritmy, které jsem popsal v teoretické části má uživatel k dispozici a může je v programu libovolně používat.

5.1 Použité technologie

Celý zdrojový kód je koncipován jako knihovna. K vývoji bylo použito vývojové prostředí *Visual Studio*. Celá knihovna byla napsána v programovacím jazyce C# verze 9.0 [7]. Jazyk C# je vysokoúrovňový objektově orientovaný jazyk. Obě tyto komponenty zastřešuje technologie .NET, která je vyvíjena a spravována společností Microsoft. Celá knihovna je postavena na .NET verzi 5.0. [8]

5.2 Konečné automaty

Implementace konečných automatů je postavena na pěti třídách.

Třída *State*, která v sobě uchovává *id*, pojmenování stavu a informaci o tom, zda je stav počáteční, přijímající nebo obyčejný.

```
public class State
    {
        public int Id { get; init; }

        private string mLabel;
        public bool IsInitial { get; internal set; }
        public bool IsAccept { get; internal set; }

        public string Label
        {
            get
```

```
            {
                if (string.IsNullOrEmpty(mLabel))
                    return Id.ToString();
                return mLabel;
            }
            init
            {
                mLabel = value;
            }
        }
    }
```

Listing 5.1: Třída *State*

Dále obsahuje dva konstruktory. První je inicializační konstruktor a je to typický konstruktor v programování, využívá se zcela běžně. Druhý konstruktor slouží k načítaní stavu ze souboru. Tento konstruktor se spouští jedině tehdy, pokud uživatel načítá automat ze souboru.

Další důležitou součástí je třída reprezentující přechod. Třída *DeltaFunctionTriplet*:

```
public class DeltaFunctionTriplet
    {
      public int From { get; }
      public char By { get; }
      public int To { get; }
   }
```

Listing 5.2: Třída *DeltaFunctionTriplet*

Třída obsahuje mimo vlastností (tzv. **Property**) pouze jeden inicializační konstruktor.

Samostatná implementace automatů začíná až v třídě *AbstractFiniteAutomaton*. Její role ve zdrojovém kódu je role **rodiče**, ze které pak dále dědí třídy *DeterministicFiniteAutomaton* a *NondeterministicFiniteAutomaton*. Třída obsahuje **dictionary**, což je v jazyce C# možná implementace pro hashovací tabulku. V tomto **dictionary** je klíčem **id** stavu a hodnotou samotná instance třídy *State*. Obsahem této třídy je také seznam, který je určený pouze pro čtení. Seznam v sobě uchovává takové instance třídy *State*, které jsou označeny jako přijímající. Ve třídě je uložen řetězec, který reprezentuje abecedu. Vyhodnotil jsem, že není potřeba abecedu definovat jiným způsobem, než řetězcem, protože program s abecedou pracuje pouze na základní úrovni, to znamená, že potřebuje maximálně v nějakých algoritmech pracovat s jednotlivými znaky abecedy a **string**, jakožto datový typ pro řetězec tuto práci umožňuje. Uživatel může s touto třídou pracovat pouze prostřednictvím tříd *DeterministicFiniteAutomaton* a *NondeterministicFiniteAutomaton*.

```
public class AbstractFiniteAutomaton
    {
      protected Dictionary<int, State> States = new Dictionary<int, State>();

      protected string Alphabet;

      public IReadOnlyList<State> AcceptStates
      {
         get
         {
            List<State> f = new List<State>();
            foreach (State s in States.Values)
            {
               if (s.IsAccept)
               {
                  f.Add(s);
               }
            }
            return f;
         }
      }
   }
```

Listing 5.3: Třída *AbstractFiniteAutomaon* bez funkcí

Všechno, co mají deterministické i nedeterministické automaty společné již máme zadefinováno. Nyní potřebujeme vytvořit třídu pro deterministický automat *DeterministicFiniteAutomaton*, která dědí ze třídy *AbstractFiniteAutomaton*. Obsahem třídy je celočíselná proměnná, která značí jeden počáteční stav. Dále potřebujeme reprezentovat přechodovou funkci. Pro jeho reprezentaci jsem zvolil **dictionary**, a to z důvodu přístupu. Klíčem **dictionary** je *id* výchozího stavu a hodnotou je **sorted list**, jehož klíčem je symbol, značící přechod a hodnota je *id* stavu, do kterého přechod vede. Díky tomuto způsobu je program schopen efektivně realizovat přechodovou funkci.

```
public class DeterministicFiniteAutomaton : AbstractFiniteAutomaton
    {
        protected Dictionary<int, SortedList<char, int>> DeltaFunction = new
            Dictionary<int, SortedList<char, int>>();
```

```
        protected int InitialStateId;
    }
```

Listing 5.4: Třída *DeterministicFiniteAutomaton* bez funkcí

Instanci této třídy může uživatel načíst ze souboru, nebo převést z nedeterministického konečného automatu pomocí funkce *ConvertToDeterministicFiniteAutomaton*, viz. kapitola 5.5.

Poslední a velmi důležitou částí pro realizaci automatů ve zdrojovém kódu je třída *NondeterministicFiniteAutomaton*, která také dědí ze třídy *AbstractFiniteAutomaton*. Narozdíl od deterministických automatů má tato třída proměnnou pro seznam více *id*, protože nedeterministické automaty mohou mít počátečních stavů více. Přechodová funkce je realizována pomocí **dictionary**, s tím rozdílem, že hodnotou setříděného seznamu (**sorted list**) není jedna celočíselná proměnná pro jeden stav, nýbrž seznam. Třída disponuje ještě jedním **dictionary**, jehož klíčem je celočíselná proměnná pro jedno *id*. Hodnotou je taktéž seznam *id*. Tento seznam v sobě uchovává informace o ε-přechodech.

```
public class DeterministicFiniteAutomaton : AbstractFiniteAutomaton
    {
        protected List<int> InitialStateIds;

        protected Dictionary<int, SortedList<char, List<int>>> DeltaFunction = new
            Dictionary<int, SortedList<char, List<int>>>();

        protected Dictionary<int, List<int>> EpsilonDeltaFunction;
    }
```

Listing 5.5: Třída *NondeterministicFiniteAutomaton* bez funkcí

Instanci této třídy lze také načíst ze souboru. Uživatel si může vytvořit zcela vlastní instanci této třídy pomocí čtyřparametrového konstruktoru. Do konstruktoru musí zadat seznam stavů, uchovávající instance třídy *State*. Dále musí program znát řetězec jako abecedu. Také potřebuje seznam přechodů, takže seznam instací třídy *DeltaFunctionTriplet* a v poslední řadě také seznam, přesněji sorted list, pro ε-přechody. Poslední dva parametry mohou být prázdné seznamy.

```
NondeterministicFiniteAutomaton NFA = new NondeterministicFiniteAutomaton(states,
    Alphabet, dft, EpsilonTransition);
```

Listing 5.6: Vytvoření instance třídy *NondeterministicFiniteAutomaton* uživatelem

5.2.1 Algoritmus realizující výpočet automatu

Tento algoritmus realizuje funkce *Accepts*. Na vstupu očekává řetězec, který reprezentuje slovo z abecedy. Funkce vrací logickou hodnotu na základě toho, zda automat dané slovo přijímá, či nikoliv.

U deterministických automatů je algoritmus celkem jednoduchý. Funkce příjme řetězec. Tento řetězec čte znak po znaku a po přečtení každého znaku zavolá funkci *DoDeltaFunction*, která nastavuje řídící jednotku na aktuální stav. Po přečtení všech znaků vrátí funkce logickou hodnotu **true** nebo **false**.

```
public bool Accepts(string input)
        {

            int currenentStateId = this.InitialStateId;
            foreach (char c in input)
            {
                currenentStateId = DoDeltaFunction(currenentStateId, c);
            }

            return AcceptStates.Any(x => x.Id == currenentStateId);
        }
```

Listing 5.7: Funkce *Accepts* u deterministických automatů

Nedeterministické automaty mají algoritmus podobný, avšak trochu složitější. Na vstupu funkce se opět nachází slovo ve formě řetězce. Na toto slovo se zavolá funkce *ExpandCurrentStatesByEpsilonTransitions*, která rozšíří seznam počátečních stavů i o stavy, do kterých se řídící jednotka dostane pomocí ε-přechodů. Dále algoritmus čte znak po znaku a postupně posouvá řídící jednotku seznamem aktuálních stavů. Pokud se algoritmus dostane do stavu, ze kterého vede ε-přechod, zavolá se funkce *GoThroughEpsilon*, která zaručí, aby se aktuálním stavem stal stav, do kterého ε-přechod vede. Původní stav však již nebude na seznamu aktuálních stavů. Po přečtení všech znaků se opět zavolá funkce *ExpandCurrentStatesByEpsilonTransitions* pro rozšíření seznamu aktuálních stavů stavy, do kterých se lze dostat pomocí ε-přechodů. Funkce vrátí logickou hodnotu, na základě řídící jednotky. Pokud se řídící jednotka nachází alespoň v jednom z přijímajících stavů, tak funkce vrátí **true**, v opačném případě vrací **false**.

```
public bool Accepts(string input)
        {
            List<int> currentStateIds = new List<int>(InitialStateIds);
```

```
        ExpandCurrentStatesByEpsilonTransitions(currentStateIds);

        foreach (char c in input)
        {
            while (HasEpsilonTransition(currentStateIds))
            {
                GoThroughEpsilon(currentStateIds);
            }
            currentStateIds = DoDeltaFunction(currentStateIds, c);
            if (currentStateIds.Count == 0)
            {
                return false;
            }
        }

        ExpandCurrentStatesByEpsilonTransitions(currentStateIds);

        return AcceptStates.Any(x => currentStateIds.Contains(x.Id));
    }
```

Listing 5.8: Funkce *Accepts* u nedeterministických automatů

5.3 Práce se soubory

Uživatel má možnost deterministické i nedeterministické automaty uložit. Ty může také zpátky načíst ze souboru, pokud zná k nim cestu.

K práci se soubory jsem zvolil načítání a ukládání do XML souborů. Vzhledem k tomu, že automaty mohou být značně rozsáhlé a mohou mít několik tisíc i milionu stavů a přechodů, tak díky XML souborům může uživatel tato data efektivně a rychle číst, protože XML soubory jsou jednoduché na čtení. Soubor obsahuje atribut **Type** elementu **Automaton**, který v sobě uchovává informaci o tom, zda je ukládaný automat deterministický nebo nedeterministický. Díky tomuto elementu program pozná, jaký typ automatu načítá, respektive ukládá.

```
NondeterministicFiniteAutomaton NFA = new NondeterministicFiniteAutomaton(states,
    Alphabet, dft, EpsilonTransition);
NFA.Save2Xml("test.xml");

NondeterministicFiniteAutomaton NFA2 = XmlAutomataReader.ReadFromXml("test.xml");
```

Listing 5.9: Ukázka ukládání a načítání ze souboru

5.3.1 Ukládání automatů do souboru

Pro ukládání automatů do souborů slouží funkce *Save2Xml*. Tato funkce je součástí tříd *DeterministicFiniteAutomaton* i *NondeterministicFiniteAutomaton*. Vstupním parametrem funkce je řetězec, reprezentující cestu, kam se má soubor uložit. Funkce nejdříve vytvoří element **Automaton**, kterému nastaví atribut na *deterministic* nebo *nondeterministic*, podle typu ukládaného automatu. Dále zavolá stejnojmennou funkci ze svého předka, tedy ze třídy *AbstractFiniteAutomaton*. Funkce z předka má za úkol nejdříve do souboru uložit stavy a abecedu. Poté se funkce vrací zpátky a ukládá přechodovou funkci, případně i ε přechodové funkce, pokud se v automatu nachází.

5.3.2 Načítání automatů do souboru

Funkce pod názvem *LoadFromXml* přijímá na vstupu řetězěc, tedy cestu k souboru. Funkce pro načtení automatu nejdříve načte atribut **Type** a zkontroluje, zda se načítá správný typ automatu. Například v případě načítaného deterministického automatu tedy program zkontroluje, zda nenačítá nedeterministický, pokud ano, funkci ukončí a vypíše zprávu do konzole.

Pomocí třídy *XMLNode*, která je součástí balíčku *System.Xml*, postupně načítá jednotlivé části souboru a naplňuje seznamy určené pro konstruktor pro vytvoření automatu. Z těchto seznamů poté vytvoří nový objekt třídy, který funkce vrátí.

5.4 Redukce automatu

Redukce automatu se dělí na tři funkce, které se spouštějí v určeném pořadí.

5.4.1 Odstranění nedosažitelných stavů – funkce DeleteUnattainableStates

Funkce je součástí třídy *NondeterministicFiniteAutomaton*.

Na začátku funkce jsou definované dva seznamy. Jeden pro dosažitelné stavy a jeden pro aktuálně zpracovávané stavy (ekvivalent pro řídící jednotku). Oba seznamy se zpočátku naplní všemi počátečními stavy. Pomocí funkce *ExpandAttainableStates*, která se nachází v cyklu, program postupně rozšiřuje seznam dosažitelných stavů a zajišťuje novou inicializaci pro právě zpracovávané stavy. Cyklus běží do doby, dokud není seznam pro aktuálně zpracovávané stavy prázdný. Po ukončení cyklu vytvoří algoritmus dvě proměnné typu **dictionary**. Tyto nově vytvořené proměnné se naplní přechodovými funkcemi a ε přechodovými funkcemi ze všech dosažitelných stavů. Nakonec tyto nové

proměnné **dictionary** nahradí původní proměnné pro přechodovou a ε přechodovou funkci.V **dictionary** stavů, obsažené ve třídě *AbstractFiniteAutomaton* se smažou všechny stavy, které nejsou na seznamu dosažitelných stavů.

Algoritmus 1: Algoritmus na odstranění nedosažitelných stavů

```
currentStates := initialStates;
attainableStates := initialStates;
while currentStates.count != 0 do
    ExpandAttainableStates(currentStatesId, attainableStatesId);
end
```

5.4.2 Odstranění nadbytečných stavů – funkce DeleteUnnecessaryStates

Funkce je součástí třídy *NondeterministicFiniteAutomaton*.

Algoritmus této funkce funguje téměř stejně jako algoritmus pro odstranění nedosažitelných stavů s tím rozdílem, že seznam, který zpracovává aktuální stavy se zpočátku inicializuje na stavy přijímající. A pochopitelně algoritmus pracuje s funkcí *ExpandNecessaryStates* pro rozšíření seznamu stavů, které nejsou nadbytečné.

Algoritmus 2: Algoritmus na odstranění nedosažitelných stavů

```
foreach state in acceptStates do
    necessaryStates.Add(state);
    currentStatesIds.Add(state);
end
while currentStates.count != 0 do
    ExpandNecessaryStates(currentStatesIds, necessaryStatesIds);
end
```

5.4.3 Odstranění ekvivalentních stavů – funkce DeleteEquivalentStates

Funkce je součástí třídy *DeterministicFiniteAutomaton*, protože algoritmus na odstranění ekvivalentních stavů v nedeterministických automatech nejspíš ani neexistuje.

V teoretické části této bakalářské práce jsem uvedl, jakým způsobem odstranit ekvivalentní stavy, a to pomocí tabulky viz. kapitola 2.4.4 . V programu jsem si proto vytvořil proměnnou typu **dctionary** jako alternativu tabulky. Klíčem je celé číslo, které reprezentuje *id* stavu, hodnotou je potom setříděný seznam (**sorted list**), jehož klíčem je symbol, značící přechod, a hodnota je celé číslo, *id* stavu, do kterého se pomocí daného přechodu dostaneme.

Dále jsem nadeklaroval další **dictionary**, pojmenované *GroupRecorder*. Klíčem tohoto **dictionary** je opět celé číslo a značí pomyslné *id* množiny. Hodnotou je seznam celých čísel a ten nám udává všechna *id* stavů v dané množině.

Díky těmto dvěma proměnným je program schopný vyhodnotit do jaké množiny patří jaký stav.

Nejdříve si program všechny stavy rozdělí do dvou množin. První množinu naplní přijímajícími stavy a druhou nepřijímajícími. Poté začíná cyklus. První funkce cyklu *EditTable* vyplní tabulku. Následují další tři funkce *FindDifferentStates*, *DeleteDifferentStatesFromGroupRecorder* a *ClassifyDifferentStatesIntoRightGroups*. Tyto funkce zajistí, aby se všechny stavy z tabulky dostaly do správných množin, popřípadě vytvořili novou množinu. Tvorba nové množiny se rozhodne na základě seznamu *differentStates*. Pokud není seznam po průběhu čtyř zmiňovaných funkcí prázdný, algoritmus ví, že se musí vytvořit nová množina stavů. Pokud se vytvoří nová množina, tak cyklus proběhne znova do doby, dokud algoritmus neprojde alespoň jednu iteraci cyklu bez vytvoření nové množiny. Po ukončení cyklu by měly být všechny stavy ve správných množinách a nyní je potřeba vybrat jen jeden stav z každé množiny a vytvořit z nich nový, redukovaný automat.

Nadeklaruji si proto další dvě proměnné typu **dictionary**. V těchto dvou nových proměnných budou uložené stavy a přechodové funkce z redukovaného automatu.

Algoritmus je naprogramován tak, aby vybral vždy první stav z každé množiny. Pokud se v množině nachází počáteční stav, bude vybraný stav, reprezentující svojí množinu, také počáteční.

Algoritmus 3: Algoritmus na odstranění ekvivalentních stavů

```
foreach state in states do
    if state je přijímající then
        GroupRecorder.Add(1, state);
    else
        GroupRecorder.Add(2, state);
    end
end
groupCreated := false;
while groupCreated do
    EditTable(GroupRecorder, Table);
    FindDifferentStates(differentStates, GroupRecorder, Table);
    DeleteDifferentStatesFromGroupRecorder(differentStates, GroupRecorder, Table);
    ClassifyDifferentStatesIntoRightGroups(differentStates, GroupRecorder, Table);
    while differentStates.Count != 0 do
        groupCreated = true;
        CreateNewGroup(differentStates, GroupRecorder, Table);
    end
end
```

5.5 Převod nedeterministického automatu na deterministický

Funkce s názvem *ConvertToDeterministicFiniteAutomaton*.

Na začátku funkce je deklarovaný **dictionary** *StateRecorder*, jehož klíčem je řetězec (**string**) a hodnotou je celé číslo. Jeho úkol bude uchovávat jednotlivé unikátní řetězce, ke kterým bude algoritmus přiřazovat číslo stavu v novém deterministickém automatu.

Začátkem výpočtu algoritmu jsou všechny počáteční stavy. Díky už známé funkci *Expand-CurrentStatesByEpsilonTransitions* zahrne k nim i stavy, po kterých se počáteční stavy dostanou po ε přechodech, a vytvoří seznam aktuálně zpracovávaných stavů. Řekněme, že v tomto seznamu budou uložené tři stavy, stavy 1, 2 a 3. V tento okamžik se vytvoří jeden řetězec z těchto stavů. Vytvoření řetězce zaručuje funkce *BuildStateId* a z těchto tří stavů vytvoří řetězec **1+2+3** a ten se pak zapíše do *StateRecorder*. Vzhledem k tomu, že v *StateRecorder* bude tento řetězec nový, tak se k němu vytvoří nové *id*. Za předpokladu, že by *StateRecorder* již tento řetězec obsahoval, tak se do něj nepřidá. Algoritmus poté vezme přidaný řetězec, zpátky ho rozdělí na *id* čísla stavů 1,2 a 3. U všech těchto stavů provede přechodovou funkci, včetně ε přechodů a všechny dosažené stavy zapíše do seznamu aktuálně zpracovávaných stavů. Tento seznam opět převede na řetezec pomocí funkce *BuildStateId* a zapíše jej do *StateRecorder*. Algoritmus takhle zpracuje úplně všechny řetězce v *StateRecorder* a provede všechny přechodové funkce. Po skončení tohoto algoritmu vytvoří novou instanci třídy *DeterministicFiniteAutomaton*.

Existují stavy, které nemají definovanou přechodovou funkci přes symbol z abecedy, v takovém případě algoritmus vytvoří speciální stav, nastaví mu *id* 0 a pojmenuje jej symbolem $\emptyset$. Tomuto speciálnímu stavu vytvoří přechodové funkce přes všechny symboly z abecedy, a ty povedou do tohoto stavu, takže bude tento stav zacyklený sám na sebe.

Algoritmus 4: Algoritmus na převod na deterministický konečný automat

```
currentStates := initialStates;
stateCounter := 0;
for i ; i < stateCounter ; i++ do
    ExpandCurrentStatesByEpsilonTransitions(currentStateIds);
    string keyStateId = BuildStateId(currentStates);
    foreach symbol in alphabet do
        foreach state in currentStates do
            EndStates = DeltaFunction[state];
            ExpandCurrentStatesByEpsilonTransitions(EndStates);
        end
        string valueStateId = BuildStateId(EndStates);
        StateRecorder.Add(valueStateId);
        stateCounter++;
        NewDFT.Add(StateRecorder[keyStateId], symbol, StateRecorder[valueStateId]);
    end
end
```

5.6 Odstranění epsilon přechodů

Funkce se ve zdrojovém kódu nachází pod názvem *DeleteEpsilonTransitions*.

Na začátku funkce je deklarovaný **dictionary** *EpsilonSeals*. Jeho účel je zaznamenávat ε-uzávěry všech stavů. Také je deklarovaný další **dictionary** s názvem *NewDeltaFunction*, kde se budou ukládat nové přechodové funkce. Algoritmus zaznamená ke všem stavům jeho ε-uzávěry. V teoretické části této bakalářské práce, kapitole 2.6, jsem uvedl, že ve svém ε-uzávěru jsou i stavy samotné, nicméně program ukládá do ε-uzávěrů pouze stavy navíc, kromě sebe samotných.

V dalším kroku ke všem počátečním stavům přiřadíme status počátečního ke všem stavům v jeho uzávěru.

Dále pomocí funkce *FindLinkedStates*, najdeme ke každému výchozímu stavu S stavy, spojené ε-přechodem. Ke každému takovému stavu povedeme nový přechod, pro každý přechod, který vede do výchozího stavu S. Každý nový přechod zaznamenáváme do proměnné *NewDeltaFunction*.

Zbývají už poslední dva kroky algoritmu. Nejdříve doplní původní přechodovou funkci, proměnnou *DeltaFunction*, o nové přechodové funkce z proměnné *NewDeltaFunction*. Jako poslední krok vymaže všechny ε-přechody, to znamená, že pouze vymaže celý obsah proměnné *EpsilonDeltaFunction*.

Algoritmus 5: Algoritmus na odstranění ε přechodů

```
foreach state in states do
    currentStatesIds.Add(state);
    ExpandCurrentStatesByEpsilonTransitions(currentStatesIds);
    EpsilonSeals.Add(state, new List<int>(currentStatesIds));
end
foreach epsilonSeal in epsilonSeals do
    if epsilonSeal obsahuje počáteční stav then
        foreach state in epsilonSeal do
            state := initial;
        end
    end
end
foreach epsilonSeal in epsilonSeals do
    LinkedStates = findLinkedStates(epsilonSeal);
    foreach linkedState in LinkedStates do
        foreach state in epsilonSeal do
            NewDeltaFunction[LinkedState].Add(state);
        end
    end
end
EpsilonDeltaFunction.Clear();
```

5.7 Převod regulárních jazyků na konečný automat

Konečný automat se dá v programu vytvořit nejen prostřednictvím ručního vytvoření instance třídy *NondeterministicFiniteAutomaton*, nebo načtením ze souboru, ale dá se definovat také pomocí regulárních jazyků, přesněji regulárním výrazem, regulární gramatikou a derivací regulárního výrazu. Pro všechny případy je v programu definovaná třída *AutomataBuilder*, která obsahuje několik metod, díky kterým je možné z regulárních jazyků vytvořit automat, přesněji nedeterministický automat.

5.7.1 Převod regulárního výrazu na konečný automat

Aby mohl program převést RV na ZNKA musí si nejdříve RV patřičně rozdělit na části, a z těch poté sestavovat menší automaty, a ty nakonec spojit ve finální konečný automat. [6]

5.7.1.1 Třída TreeNode

Začněme třídou *TreeNode*. Tato jednoduchá třída se skládá z řetězce (**Label**) a z pole (**Children**) typu TreeNode, do kterého se bude tato třída rekurzivně ukládat. Podle počtu prvků v rekurzivně uloženém poli poté algoritmus pozná, o jakou část RV se jedná a bude vědět, jakou konstrukci automatu vytvořit.

```
internal class TreeNode
    {
        public string Label { get; init; }

        public TreeNode [] Children { get; init; }

        internal TreeNode(string Label, TreeNode[] Children)
        {
            this.Label = Label;
            this.Children = Children;
        }
    }
```

Listing 5.10: Třída *TreeNode*

5.7.1.2 Třída RegularExpressionParser

V této bakalářské práci jsme si již definovali formální gramatiku v kapitole 4.1, kterou program využívá k rozdělování RV na části. Ke každému neterminálu z formální gramatiky je vytvořena funkce, která vrací instanci třídy *TreeNode*.

Nicméně pokud chceme začít rozdělovat RV a dostat výslednou instanci třídy *TreeNode*, musíme zavolat funkci *toParseTree*, která je součástí třída *AutomataBuilder*. Ta spustí funkci pro neterminál **Expression**. Od této chvíle se začnou ostatní funkce zbylých neterminnálu rekurzivně volat a ukládat data do *TreeNode*, kterou pak funkce *toParseTree* vrátí.

```
public TreeNode toParseTree()
      {
          return Expression();
      }
```

Listing 5.11: Funkce *toParseTree*

5.7.1.3 Třída AutomataBuilder

Třída obsahuje funkce pro vytvoření základních automatů. Například funkci *FromSymbol*, která vytvoří dva stavy a vytvoří k nim přechod, přes daný symbol. Obdobně obsahuje další funkce na vytvoření sjednocení automatů, zřetězení automatů, nebo iterace automatů, tak jak bylo definováno pro sekci regulárních výrazů v kapitole 3.

Důležitou funkcí je *toNFAfromParseTree*, která na vstupu přijímá instanci třídy *TreeNode*. Tato funkce je složena pouze z podmínek, aby zjistila, kterou část RV (například **term** nebo **atom**) zpracovává. Také se rekurzivně volá na základě rekurze třídy *TreeNode* a postupně volá funkce pro sjednocení, zřetězení atd. tak, jak je potřeba, a zároveň se tyto funkce samy postarají i o spojení menších částí automatu do jednoho výsledného automatu. Výstupem funkce je nová instance nedeterministického automatu.

Pokud uživatel chce zkonstruovat automat převodem z RV, musí zavolat funkci *BuildAutomatonFromRegularExpression*, která je součástí třídy *AutomataBuilder*. Funkce přijímá dva parametry. První je regulární výraz ve formě řetězce, druhým parametrem je abeceda, taktéž formou řetězce. Funkce nejdříve vytvoří instanci třídy *RegularExpressionParser*, do které uloží konkrétní RV, třída tento RV rozdělí na části a vrátí instanci třídy *TreeNode*. Poté pouze zavolá funkci *toNFAfromParseTree*, kde na vstupu bude instance třídy *TreeNode*. Funkce *toNFAfromParseTree*, vrátí už vytvořený automat, který poté vrátí i samotná funkce *BuildAutomatonFromRegularExpression*.

```
string RegEx = "a?b";
NondeterministicFiniteAutomaton NFA = builder.
    BuildAutomatonFromRegularExpression(RegEx, Alphabet);
```

Listing 5.12: volání funkce *BuildAutomatonFromRegularExpression*

5.7.2 Převod derivace regulárních výrazů na konečný automat

Tento převod zahrnuje jen jedna funkce *BuildAutomatonFronDerivationOfRegularExpression*, a to ve třídě *AutomataBuilder*. Na vstupu funkce musí být definována množina řetězců. Funkce nejdříve vytvoří jeden počáteční stav. Poté přečte každý vstupní řetězec v seznamu řetězců a vytvoří k nim ekvivalentně stavy a přechody. Jeden symbol z řetězce znamená nový stav a přechod, přes daný symbol, vedoucí do nového stavu. Až algoritmus přečte všechny symboly, tak označí každý posledně přidaný stav každého automatu jako přijímající. Ke všem takto vytvořeným automatům z jednotlivých řetězců vede ε přechod z prvního počátečního stavu do všech prvních stavů každého automatu.

5.7.3 Převod regulární gramatiky na konečný automat

Posledním možným převodem z regulárních jazyků na KA je převod z regulární gramatiky. Takový algoritmus zahrnuje funkce *BuildAutomatonFromRegularGrammar* ve třídě *AutomataBuilder*.

Funkce přijímá seznam řetězců, které reprezentují regulární gramatiku v BNF notaci. Prvním krokem algoritmu je vytvoření stavů ze všech neterminálů. Poté se vytvoří ještě jeden stav, který bude přijímající a nese název **final**. V cyklu se pomocí metody *Split*, která může rozdělovat řetězce na základě libovolného znaku, rozdělují řetězce regulární gramatiky na části a ty se zpracovávají. Algoritmus také hlídá počet terminálních znaků, a pokud je víc, než jeden, tak vytváří dodatečné stavy a přechody podle pravidla regulární gramatiky. Pokud se na pravé části pravidla nacházejí pouze terminály, tak algoritmus vyhodnotí, že přechod přes poslední terminální znak bude přechod vedoucí do přijímajícího stavu **final**.

5.8 Program GraphViz

GraphViz je vizualizační program, který dokáže na základě definovaného řetězce vygenerovat různé grafy, diagramy, stromy atd. Při zpracovávání této bakalářské práce jsem GraphViz využíval, abych si mohl automaty zobrazit v grafické podobě, taktéž jsem implementoval metody, které jsou součástí tříd *DeterministicFiniteAutomaton* a *NondeterministicFiniteAutomaton*. Tyto metody vracejí řetězec, který lze použít pro vygenerování grafu, resp. konečného automatu.

Aby vizualizace fungovala, potřebujeme mít program stažený a nainstalovaný v našem počítači. Dále potřebujeme vytvořit soubor s příponou **.dot*, do kterého vložíme získaný řetězec z funkcí. Program GraphViz pracuje v příkazové řádce, abychom mohli spustit obsah **.dot* souboru, musíme se v příkazové řádce dostat do složky, kde je soubor **.dot* umístěn. Poté stačí do příkazové řádky napsat příkaz: dot -Tpng *file*.dot -o *output*.png, kde *file* je název .dot souboru a *output* je název výstupního souboru ve formátu .png. Název výstupního souboru může být libovolný. Poté GraphViz vytvoří .png soubor s vygenerovaným grafem.

5.8.1 Funkce GetDotSourceCode

Funkce je součástí tříd *DeterministicFiniteAutomaton* a *NondeterministicFiniteAutomaton*. Funkce vrací řetězec, na základě kterého program GraphViz generuje automat. Obecně tato funkce v obou třídách pracuje se třídou **StringBuilder**, která umožňuje spojování a připojování řetězců. Nejdříve funkce definuje parametry, na základě kterých program GraphViz zobrazí automat, tak, jak jej zobrazit potřebujeme. Poté v cyklu do konečného řetězce zadefinujeme přijímající stavy. Poté se konečný řetězec sestavuje na základě zpracovávané třídy. Funkce dále definuje zápisem do konečného řetězce přechodové funkce a stavy, pokud se zpracovává nedeterministický automat, tak také funkce zpracovává ε přechodové funkce.

GraphViz nezná definici pro počáteční stavy a tím pádem k nim nemůže připojit šipku, která by značila počáteční stav. Proto ve funkci ke každému počátečnímu stavu definuji stav s názvem **shadow**. Ke stavu **shadow** nadefinuji přechodovou funkci přes žádný symbol, vedoucí do počátečního stavu, a původní stav **shadow** označím jako *hidden*, takže ve výstupu sice bude, ale neviditelný. Tímto způsobem realizuji všechny počáteční stavy.

Kapitola 6

Testování

Tato část bakalářské práce se zabývá testováním implementované knihovny a především demonstrací výsledků v grafické podobě. Všechny ukázky výsledných automatů budou generované programem GraphViz, který jsem popsal v kapitole 5.8.

6.1 Vytvoření nedeterministického konečného automatu

V této části si ukážeme, jak vytvořit NKA pomocí čtyrparametrového konstruktoru. Pro ukázku načtení automatu ze souboru je vyhrazena kapitola 6.4.

V programu si může uživatel vytvořit NKA pomocí konstruktoru třídy *NondeterministicFinite-Automaton*. Pro úspěšné vytvoření validního automatu je zapotřebí do konstruktoru zadat seznam stavů (třídy *State*), abecedu jako řetězec, seznam přechodových funkcí (třídy *DeltaFunctionTriplet*) a setříděný seznam, reprezentující ε přechodové funkce, jehož klíčem je celé číslo a hodnotou seznam celých čísel.

```
 string Alphabet = "ab";

List<State> states = new List<State>();
states.Add(new State(1, "q1", true, false));
states.Add(new State(2, "q2", false, false));
states.Add(new State(3, "q3", false, true));

List<DeltaFunctionTriplet> dft = new List<DeltaFunctionTriplet>();
dft.Add(new DeltaFunctionTriplet(1, 'a', 2));
dft.Add(new DeltaFunctionTriplet(1, 'a', 3));
dft.Add(new DeltaFunctionTriplet(2, 'b', 3));
```

```
SortedList<int, List<int>> epsilons = new SortedList<int, List<int>>();
epsilons.Add(1, new List<int> { 3 });

NondeterministicFiniteAutomaton NKA = new
    NondeterministicFiniteAutomaton(states, Alphabet, dft, epsilons);
```

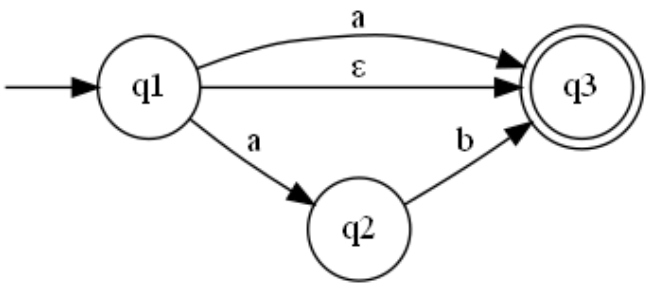
Listing 6.1: Vytvoření instance třídy *NondeterministicFiniteAutomaton*

Po vytvoření si necháme vypsat řetězec pro program GraphViz.

```
string output = NKA.GetDotSourceCode();
```

Listing 6.2: Získání řetězce pro program GraphViz

Výsledný automat vytvořený program GraphViz vypadá takto:



Obrázek 6.1: Výsledný nedeterministický konečný automat

6.2 Převod na deterministický konečný automat

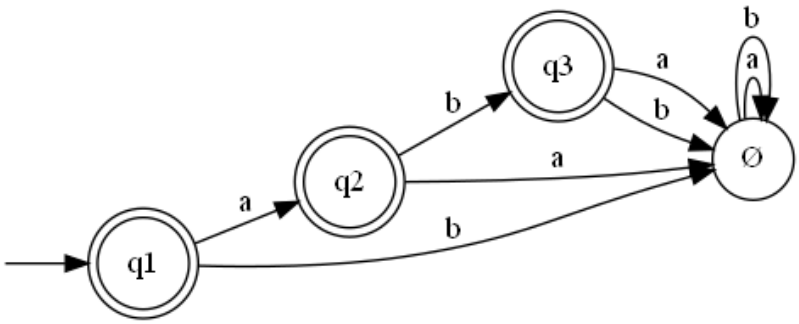
Nedeterministický konečný automat z kapitoly 6.1 převedeme na DKA. K tomu využijeme funkci *ConvertToDeterministicFiniteAutomaton*, která vrací novou instanci třídy *DeterministicFiniteAutomaton*. U výsledného DKA opět zavoláme funkci *GetDotSourceCode()*, abychom si mohi nový DKA nechat zobrazit v grafické podobě.

```
DeterministicFiniteAutomaton DKA = NKA.ConvertToDeterministicFiniteAutomaton()
    ;

string output = DKA.GetDotSourceCode();
```

Listing 6.3: Převod na DKA

Výsledný DKA:

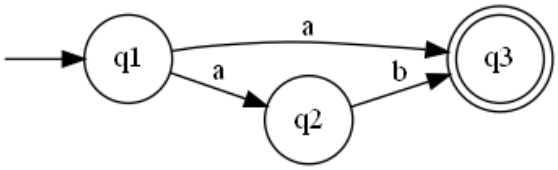


Obrázek 6.2: Výsledný deterministický konečný automat

6.3 Výpočet automatu

K demonstraci výpočtu použijeme nedeterministický automat na obrázku 6.1, pouze odstraním ε-přechod, z důvodu lepší názornosti příkladu. Funkce, která výpočet realizuje, vrací logickou hodnotu. To znamená, že v programu vytvoříme jednoduchou podmínku, která se bude ptát, zda funkce *Accepts*, realizující výpočet, vrátila logickou hodnotu **true** nebo **false**. Na základě této hodnoty vypíšeme text v konzoli aplikace. Obrázek s vypsaným výsledkem z konzole zde vložím, jako důkaz, že algoritmus funguje.

Automat, na kterém budeme funkci testovat:



Obrázek 6.3: Ukázkový automat

Program s podmínkou a volání funkce *Accepts*:

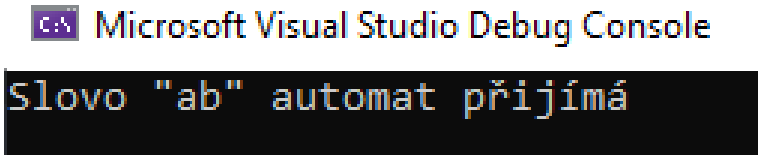
```
string slovo = "ab";

        if(NKA.Accepts(slovo))
        {
            Console.WriteLine("Slovo \""+ slovo +"\" automat přijímá");
        }
```

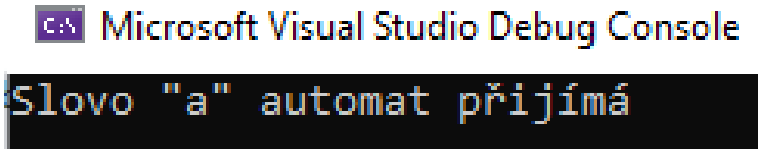
```
else
{
    Console.WriteLine("Slovo \"" + slovo + "\" automat nepřijímá");
}
```

Listing 6.4: Testování funkce *Accepts*

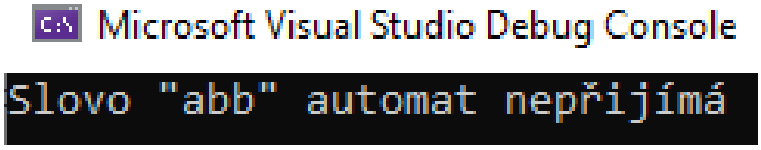
Nyní ukázka výstupů pro různá slova:



Obrázek 6.4: Výsledek pro zadané slovo: *ab*



Obrázek 6.5: Výsledek pro zadané slovo: *a*



Obrázek 6.6: Výsledek pro zadané slovo: *abb*

6.4 Práce se soubory

Pokud už máme k dispozici vytvořené automaty, můžeme je ukládat do XML souborů.

Uložení NKA na obrázku 6.1:

```
NKA.Save2Xml("NKA.xml");
```

Listing 6.5: Převod na DKA

Obsah souboru NKA.xml:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Automaton Type="Nondeterministic">
  - <States>
        <State IsInitial="True" Label="q1" Id="1"/>
        <State Label="q2" Id="2"/>
        <State Label="q3" Id="3" IsAccept="True"/>
    </States>
    <Alphabet>ab</Alphabet>
  - <DeltaFunction>
        <DeltaFunctionTriplet To="2" By="a" From="1"/>
        <DeltaFunctionTriplet To="3" By="a" From="1"/>
        <DeltaFunctionTriplet To="3" By="b" From="2"/>
    </DeltaFunction>
  - <EpsilonDeltaFunction>
        <EpsilonDeltaFunctionPair To="3" From="1"/>
    </EpsilonDeltaFunction>
</Automaton>
```

Obrázek 6.7: Obsah souboru NKA.xml

Obdobně můžeme uložit i DKA, použijeme příklad na obrázku 6.2. Obsah XML souboru vypadá pak takto:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Automaton Type="Deterministic">
  - <States>
        <State IsAccept="True" IsInitial="True" Label="q1" Id="1"/>
        <State IsAccept="True" Label="q2" Id="2"/>
        <State Label="Ø" Id="0"/>
        <State IsAccept="True" Label="q3" Id="3"/>
    </States>
    <Alphabet>ab</Alphabet>
  - <DeltaFunction>
        <DeltaFunctionTriplet To="2" By="a" From="1"/>
        <DeltaFunctionTriplet To="0" By="b" From="1"/>
        <DeltaFunctionTriplet To="0" By="a" From="2"/>
        <DeltaFunctionTriplet To="3" By="b" From="2"/>
        <DeltaFunctionTriplet To="0" By="a" From="3"/>
        <DeltaFunctionTriplet To="0" By="b" From="3"/>
        <DeltaFunctionTriplet To="0" By="a" From="0"/>
        <DeltaFunctionTriplet To="0" By="b" From="0"/>
    </DeltaFunction>
</Automaton>
```

Obrázek 6.8: Obsah souboru DKA.xml

Oba automaty můžeme i ze souborů načíst následujícími příkazy:

```
NondeterministicFiniteAutomaton NKA2 = NondeterministicFiniteAutomaton.
    LoadFromXml("NKA.xml");
```

```
DeterministicFiniteAutomaton DKA2 = DeterministicFiniteAutomaton.LoadFromXML("
    DKA.xml");
```

Listing 6.6: Načtení obou automatů ze souborů

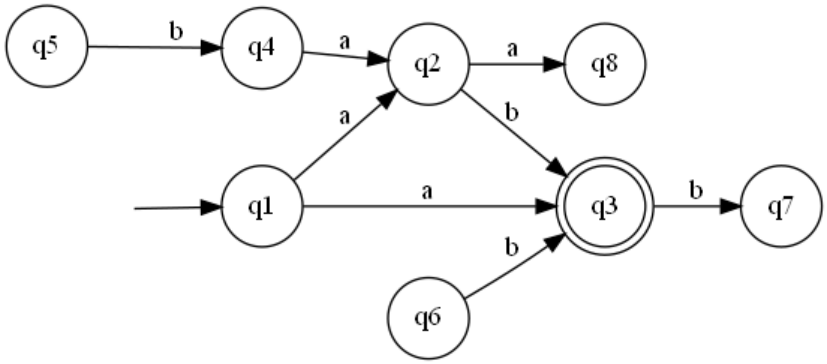
Nyní můžeme opět nechat program GraphViz vygenerovat obrázky obou automatů. Vzhledem k tomu, že tyto automaty byly vytvořeny na základě předchozích dvou automatů na obrázcích 6.1 a 6.2 a jsou tedy úplně stejné, není potřeba je zde znova zobrazovat.

6.5 Redukce automatu

V kapitole 5.4 jsme si ukázali, že se redukce automatu dělí na 3 algoritmy. Všechny tři budu postupně v této kapitole demonstrovat.

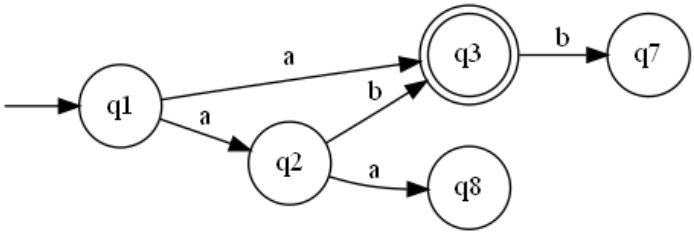
6.5.1 Odstranění nedosažitelných stavů

Ve třídě *NondeterministicFiniteAutomaton* je funkce *ReduceAutomaton*, která nejdříve provede algoritmus pro odstranění nedosažitelných stavů a poté odstranění nadbytečných stavů. Pro ukázku algoritmu na odstranění nedosažitelných stavů byl vytvořen nový NKA:



Obrázek 6.9: Příkladový automat pro odstranění nedosažitelných stavů

Po odstranění nedosažitelných stavů vypadá automat takto:



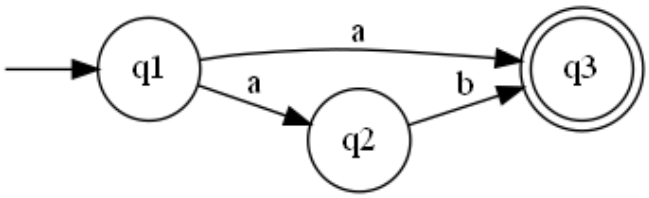
Obrázek 6.10: Příkladový automat po odstranění nedosažitelných stavů

Stavy q_4, q_5 a q_6 byly správně algoritmem vyhodnoceny jako nedosažitelné, a proto byly z automatu odstraněny.

6.5.2 Odstranění nadbytečných stavů

Algoritmus z automatu odstranil nedosažitelné stavy, nyní odstraní stavy nadbytečné. Použije k tomu automat na obrázku 6.10.

Program zavolá funkci *DeleteUnnecessaryStates*, která má na starost odstranění nadbytečných stavů. Po provedení této funkce vypadá výsledný automat takto:



Obrázek 6.11: Příkladový automat po odstranění nedosažitelných stavů

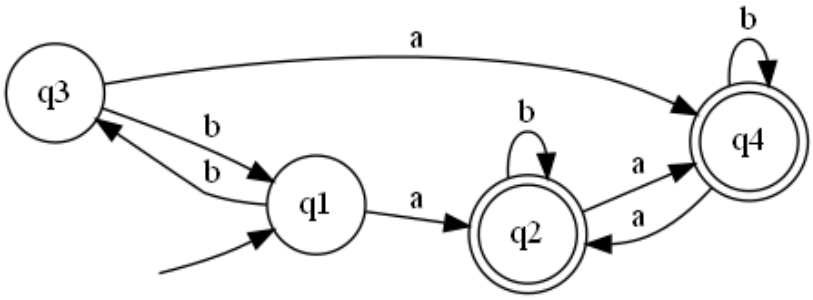
Algoritmus smazal stavy q_7 a q_8, které byly správně vyhodnocené jako nadbytečné.

```
NKA.ReduceAutomaton();
```

Listing 6.7: Odstranění nedosažitelných a nadbytečných stavů z automatu

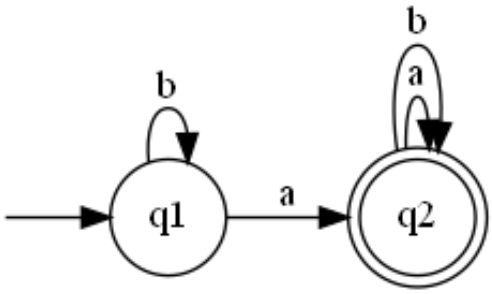
6.5.3 Odstranění ekvivalentních stavů

Tento algoritmus se dá použít pouze pro DKA, proto byl vytvořen nový příkladový automat pro lepší názornost.



Obrázek 6.12: Příkladový automat před odstraněním ekvivalentních stavů

Po odstranění ekvivalentních stavů vypadá automat následovně:



Obrázek 6.13: Příkladový automat po odstranění nedosažitelných stavů

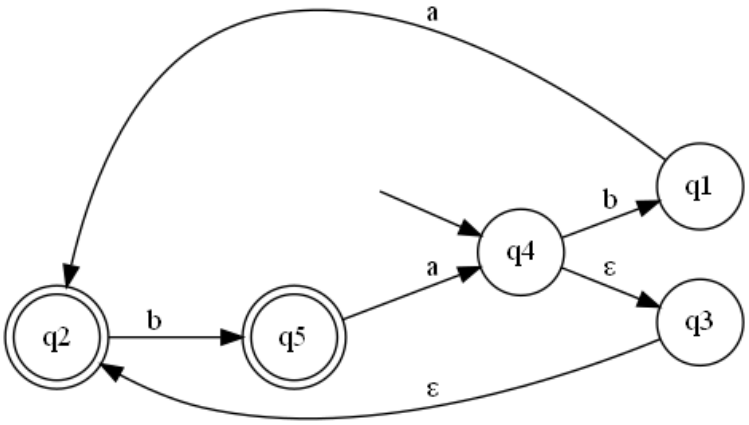
Stavy q_3 a q_4 byly správně vyhodnoceny jako ekvivalentní stavy ke stavům q_1 a q_2, a proto byly u automatu smazány.

```
DKA.DeleteEquivalentStates();
```

Listing 6.8: Odstranění ekvivalentních stavů z automatu

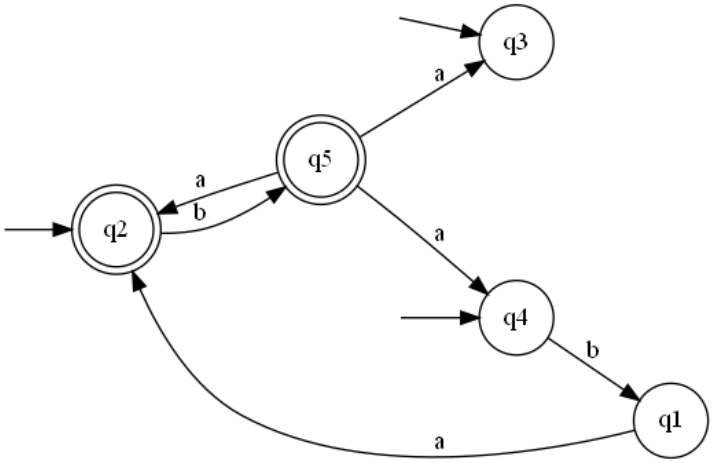
6.6 Odstranění ε přechodů

Jako ukázku použijeme automat na obrázku 2.11.



Obrázek 6.14: Příkladový automat pro odstranění ε přechodů

Ekvivalentní automat bez ε přechodů:



Obrázek 6.15: Příkladový automat po odstranění ε přechodů

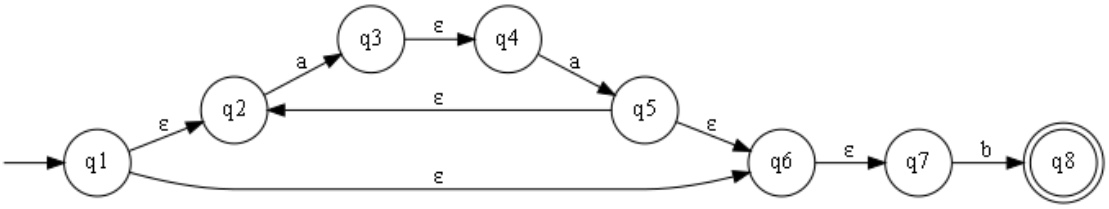
```
NKA.DeleteEpsilonTransitions();
```

Listing 6.9: Odstranění ε přechodů z automatu

6.7 Převod regulárních jazyků na konečné automaty

6.7.1 Převod regulárního výrazu na konečný automat

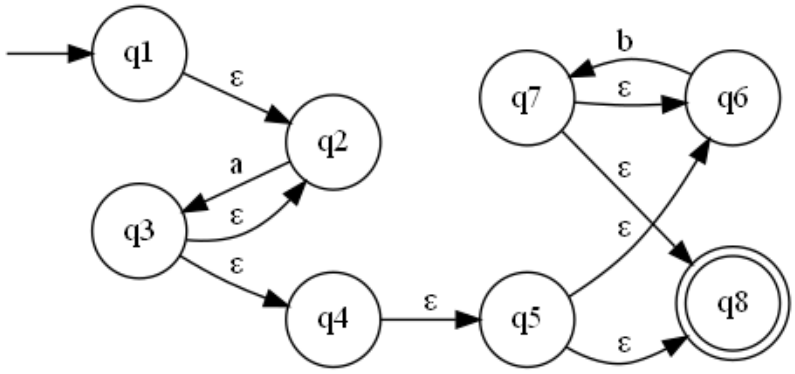
Převod z RV na ZNKA si předvedeme na několika příkladech. Funkce, která zastřešuje celý převod a vytvoření automatu přijímá na vstupu dva řetězce, a to regulární výraz a abecedu.



Obrázek 6.16: Automat sestrojený regulárním výrazem *(aa)*b*

```
string Alphabet = "ab";
string RegEx = "(aa)*b";
```

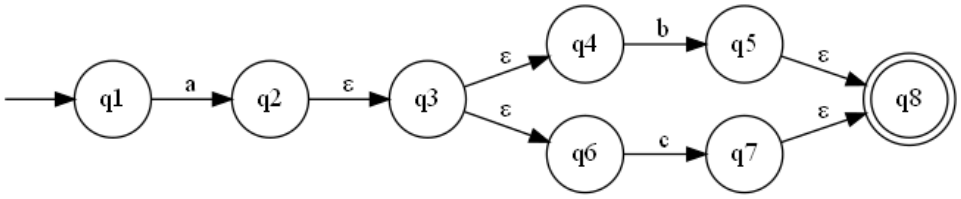
Listing 6.10: Nastavení vstupních parametrů pro funkci *BuildAutomatonFromRegularExpression*



Obrázek 6.17: Automat sestrojený regulárním výrazem *a+b**

```
string Alphabet = "ab";
string RegEx = "a+b*";
```

Listing 6.11: Nastavení vstupních parametrů pro funkci *BuildAutomatonFromRegularExpression*



Obrázek 6.18: Automat sestrojený regulárním výrazem *a(b|c)*

```
string Alphabet = "abc";
string RegEx = "a(b|c)";

AutomataBuilder builder = new AutomataBuilder();
 NondeterministicFiniteAutomaton NKA = builder.
    BuildAutomatonFromRegularExpression(RegEx, Alphabet);
```

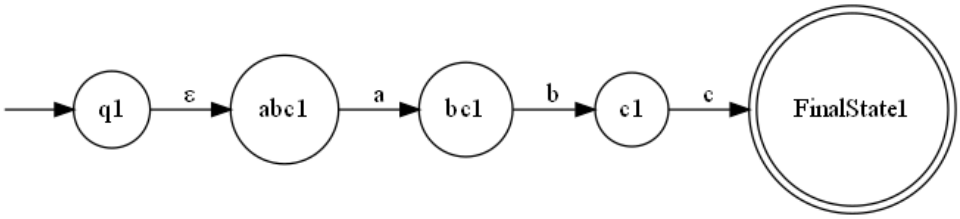
Listing 6.12: Vytváření ZNKA pomocí regulárního výrazu *a(b|c)*

6.7.2 Převod derivace regulárního výrazu na konečný automat

Funkce pro převod na KA přijímá na vstupu množinu řetězců, reprezentující slova z abecedy. Nejdřív otestujeme algoritmus pro jedno slovo:

```
HashSet<string> words = new HashSet<string>();
words.Add("abc");
```

Listing 6.13: Definice proměnné *word*

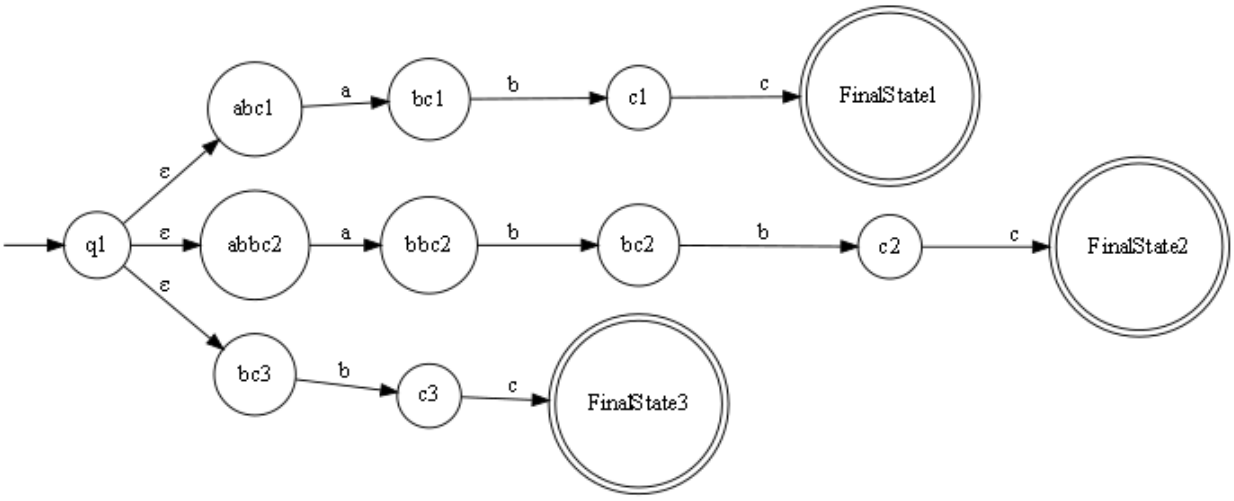


Obrázek 6.19: Automat sestrojený derivací regulárního výrazu

Poté přidáme ještě další 2 slova.

```
HashSet<string> words = new HashSet<string>();
words.Add("abc");
words.Add("abbc");
words.Add("bc");
```

Listing 6.14: Definice proměnné *words*



Obrázek 6.20: Výsledný automat po úpravě definice proměnné *words*

6.7.3 Převod regulární gramatiky na konečný automat

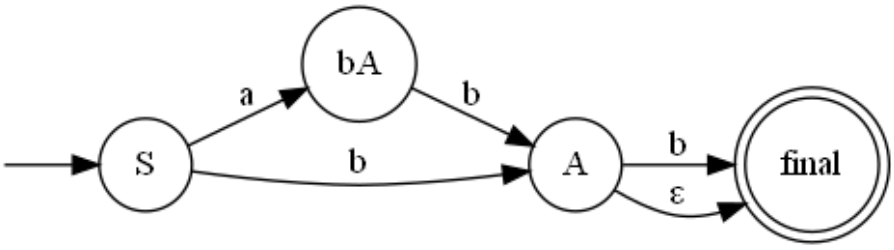
Abychom převedli regulární gramatiku na KA, musíme na vstup funkce *BuildAutomatonFromRegularGrammar* poslat seznam regulární gramatiky v BNF notaci, tak jak bylo řečeno v kapitole 4.1. Algoritmus otestujeme pro následující regulární gramatiku definovanou v programu.

```
List<string> RegularGrammar = new List<string>();

RegularGrammar.Add("<S> ::= 'ab'<A> | 'b'<A>");
RegularGrammar.Add("<A> ::= 'b' | 'ε'");
```

Listing 6.15: Definice regulární gramatiky

Výsledný automat:

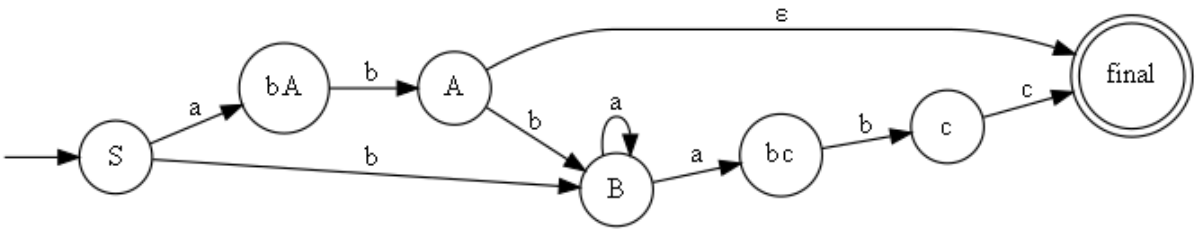


Obrázek 6.21: Výsledný automat z převedené regulární gramatiky

Regulární gramatiku ještě upravíme. První dvě pravidla trochu pozměníme a přidáme ještě i třetí pravidlo.

```
RegularGrammar.Add("<S> ::= 'ab'<A> | 'b'<B>");
RegularGrammar.Add("<A> ::= 'b'<B> | 'ε'");
RegularGrammar.Add("<B> ::= 'a'<B> | 'abc'");
```

Listing 6.16: Definice regulární gramatiky



Obrázek 6.22: Výsledný automat po úpravě regulární gramatiky

Kapitola 7

Závěr

Cílem této bakalářské práce bylo teoreticky popsat a navrhnout knihovnu funkcí pro komplexní práci s konečnými automaty. Zadání bylo splněno ve všech bodech. Doplňkem byly implementovány funkce, pomocí kterých můžeme automaty nechat zobrazit jako obrázek vizualizačním programem GraphViz. Celý program byl koncipován jako knihovna funkcí, která byla napsána v programovacím jazyce C#.První část této práce se zabývala teoretickým popisem konečných automatů, jak fungují, jejich základní druhy, a také práci s nimi.V druhé části jsme si popsali konkrétní implementaci algoritmů, které vycházeli z teoretické části.V poslední části této bakalářské práce jsme si jednotlivé algoritmy otestovali a výsledky jsme nechali zobrazit programem GraphViz.Tato práce by se dala případě rozšířit. Konkrétně pro sestavování automatů z derivace regulárních výrazů. V této práci se s derivacemi regulárních výrazů pracuje jen omezeně.Vypracování této bakalářské práce pro mě bylo přínosem v oblasti teoretických znalostí konečných automatů a regulárních jazyků.

Literatura

1. CHYTIL, Michal. *Automaty a gramatiky*. Praha: Státní nakladatelství technické literatury, 1984.
2. VACHO, Tomáš. *Algoritmy pro převod regulárních výrazů na konečné automaty* [online]. Ostrava, 2017 [cit. 2021-02-01]. Dostupné z: `https://dspace.vsb.cz/bitstream/handle/10084/119008/VAC0089_FEI_N2647_2612T025_2017.pdf?sequence=1`. Diplomová práce. Vysoká škola báňská - Technická univerzita Ostrava, Fakulta elektrotechniky a informatiky, Katedra informatiky.
3. JANČAR, Petr. *Teoretická informatika: učební text* [online]. První, 2007. Ostrava: Ediční středisko VŠB-TUO, 2007 [cit. 2021-01-31]. ISBN 978-80-248-1487-2. Dostupné z: `http://www.cs.vsb.cz/kot/download/ti2010/ti-text.2010-01-20.pdf`.
4. ČERNÍKOVÁ, Petra. *Vizualizace automatů a regulárních výrazů: Online podpora výuky* [online]. Liberec, 2012 [cit. 2021-01-31]. Dostupné z: `https://dspace.tul.cz/bitstream/handle/15240/12111/mgr_23288.pdf?sequence=1%5C&isAllowed=y`. Diplomová práce. Technická univerzita v Liberci, Fakulta mechatroniky, informatiky a mezioborových studií.
5. FLUKSA, Tomáš. *Rozhodování pravdivosti formulí Presburgerovy aritmetiky pomocí konečných automatů* [online]. Ostrava, 2010 [cit. 2021-02-02]. Dostupné z: `https://dspace.vsb.cz/bitstream/handle/10084/78952/FLU004_FEI_B2647_2612R025_2010.pdf?sequence=1`. Bakalářská práce. Vysoká škola báňská - Technická univerzita Ostrava, Fakulta elektrotechniky a informatiky, Katedra informatiky a výpočetní techniky.
6. *Parsing Regex with Recursive Descent* [online]. Sofia, Bulharsko: Denis Kyashif, 2020 [cit. 2021-04-24]. Dostupné z: `https://deniskyashif.com/2020/08/17/parsing-regex-with-recursive-descent/`.
7. TURNER, Ryan. *C#: 2 books in 1: the ultimate beginner´s & intermadiate guide to learn C# programming step by step*. [s.l]: Ryan Turner, 2018. ISBN 9781693891922.
8. *Úvod do technologie .NET* [online]. Washington: Microsoft [cit. 2021-04-26]. Dostupné z: `https://docs.microsoft.com/cs-cz/dotnet/core/introduction`.